AmpliGraph

Release 1.3.1

Luca Costabello - Accenture Labs Dublin

Mar 18, 2020

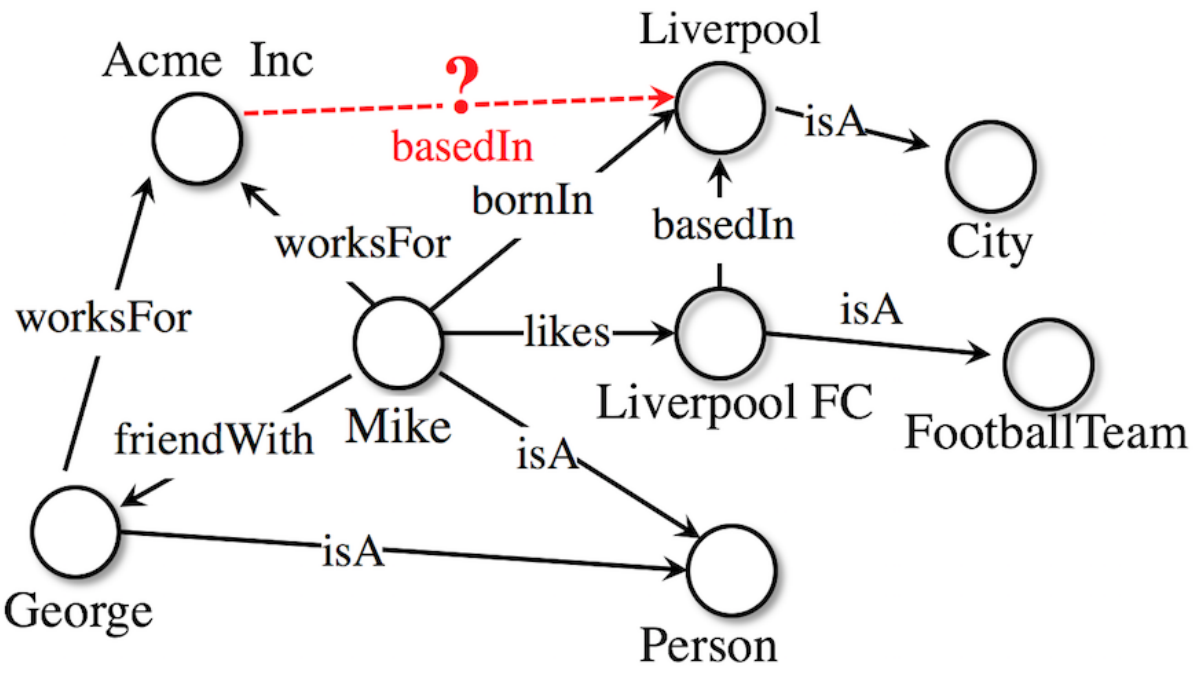
CONTENTS:

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

Go to the GitHub repository

Join the conversation on Slack

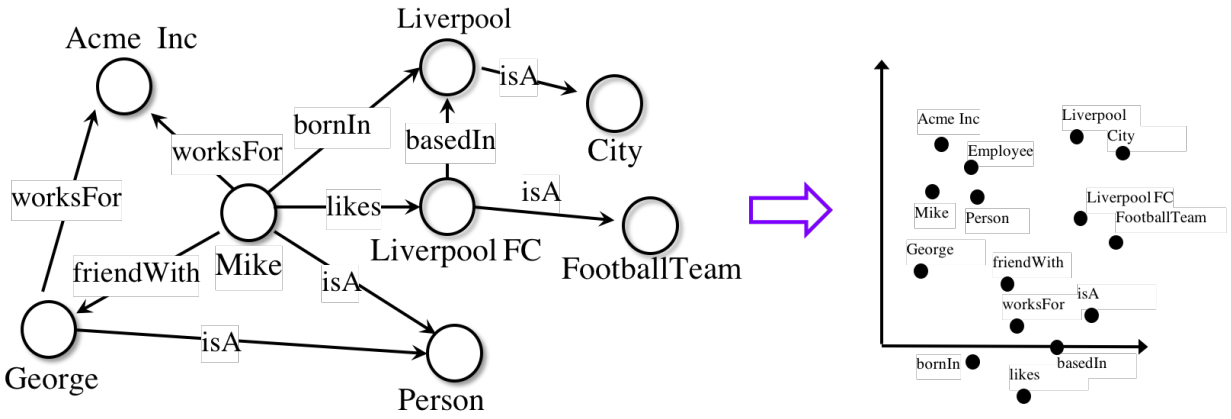
AmpliGraph is a suite of neural machine learning models for relational Learning, a branch of machine learning that deals with supervised learning on knowledge graphs.



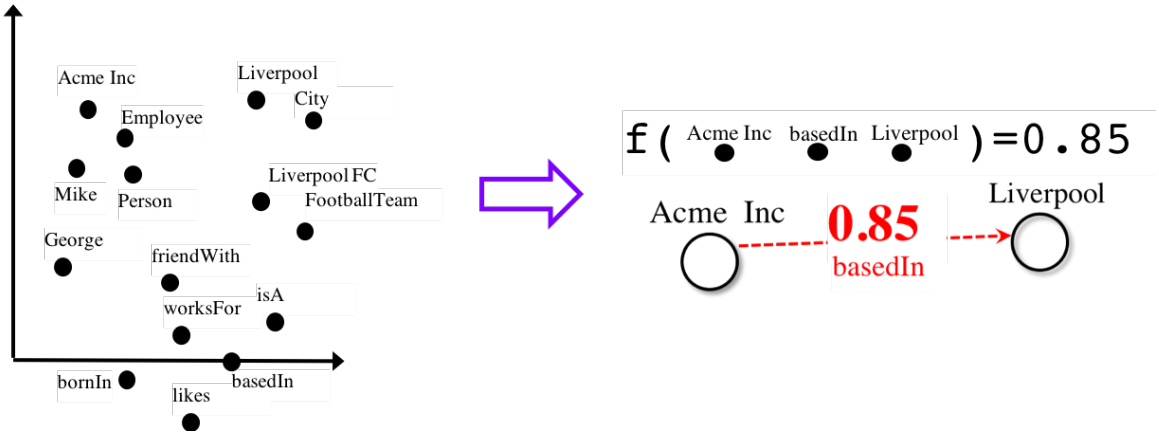
Use AmpliGraph if you need to:

- 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

ONE

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

TWO

MODULES

AmpliGraph includes the following submodules:

- **Datasets**: helper functions to load datasets (knowledge graphs).
- **Models**: knowledge graph embedding models. AmpliGraph contains **TransE**, **DistMult**, **ComplEx**, **HolE**, **ConvE**, **ConvKB** (More to come!)
- **Evaluation**: metrics and evaluation protocols to assess the predictive power of the models.
- **Discovery**: High-level convenience APIs for knowledge discovery (discover new facts, cluster entities, predict near duplicates).

CHAPTER

THREE

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 and
           Pedro Tabacof},
  title = {{AmpliGraph: a Library for Representation Learning on Knowledge Graphs}},
  month = mar,
  year  = 2019,
  doi   = {10.5281/zenodo.2595043},
  url   = {https://doi.org/10.5281/zenodo.2595043}
}
```

3.1 Installation

3.1.1 Prerequisites

- Linux, macOS, Windows
- Python 3.6

Provision a Virtual Environment

Create and activate a virtual environment (conda)

```
conda create --name ampligraph python=3.7
source activate ampligraph
```

Install TensorFlow

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

CPU-only

```
pip install "tensorflow>=1.14.0,<2.0"

or

conda install tensorflow'>=1.14.0,<2.0.0'
```

GPU support

```
pip install "tensorflow-gpu>=1.14.0,<2.0"

or

conda install tensorflow-gpu'>=1.14.0,<2.0.0'
```

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 as below (also see the *How to Contribute guide* for details):

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

3.1.3 Sanity Check

```
>> import ampligraph
>> ampligraph.__version__
'1.3.1'
```

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.

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
```

Benchmark Datasets 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_fb15k_237([check_md5hash, ...])	Load the FB15k-237 dataset
load_wn18rr([check_md5hash, clean_unseen, ...])	Load the WN18RR dataset
load_yago3_10([check_md5hash, clean_unseen, ...])	Load the YAGO3-10 dataset
load_fb15k([check_md5hash, add_reciprocal_rels])	Load the FB15k dataset
load_wn18([check_md5hash, add_reciprocal_rels])	Load the WN18 dataset
load_wn11([check_md5hash, clean_unseen, ...])	Load the WordNet11 (WN11) dataset
load_fb13([check_md5hash, clean_unseen, ...])	Load the Freebase13 (FB13) dataset

load_fb15k_237

ampligraph.datasets.**load_fb15k_237**(*check_md5hash=False*, *clean_unseen=True*, *add_reciprocal_rels=False*)

Load the FB15k-237 dataset

FB15k-237 is a reduced version of FB15K. It was first proposed by [TCP+15].

The FB15k-237 dataset is loaded from file if it exists at the `AMPLIGRAPH_DATA_HOME` location. If `AMPLIGRAPH_DATA_HOME` is not set the the default `~/ampligraph_datasets` is checked.

If the dataset is not found at either location it is downloaded and placed in `AMPLIGRAPH_DATA_HOME` or `~/ampligraph_datasets`.

The dataset is divided in three splits:

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

Dataset	Train	Valid	Test	Entities	Relations
FB15K-237	272,115	17,535	20,466	14,541	237

Warning: FB15K-237's validation set contains 8 unseen entities over 9 triples. The test set has 29 unseen entities, distributed over 28 triples.

Parameters

- **check_md5hash** (*boolean*) – If `True` check the md5hash of the files. Defaults to `False`.
- **clean_unseen** (*bool*) – If `True`, filters triples in validation and test sets that include entities not present in the training set.
- **add_reciprocal_rels** (*bool*) – Flag which specifies whether to add reciprocal relations. For every <s, p, o> in the dataset this creates a corresponding triple with reciprocal relation <o, p_reciprocal, s>. (default: False).

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_wn18rr

ampligraph.datasets.**load_wn18rr**(*check_md5hash=False*, *clean_unseen=True*, *add_reciprocal_rels=False*)

Load the WN18RR dataset

The dataset is described in [DMSR18].

The WN18RR dataset is loaded from file if it exists at the `AMPLIGRAPH_DATA_HOME` location. If `AMPLIGRAPH_DATA_HOME` is not set the the default `~/ampligraph_datasets` is checked.

If the dataset is not found at either location it is downloaded and placed in `AMPLIGRAPH_DATA_HOME` or `~/ampligraph_datasets`.

It is divided in three splits:

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

Dataset	Train	Valid	Test	Entities	Relations
WN18RR	86,835	3,034	3,134	40,943	11

Warning: WN18RR's validation set contains 198 unseen entities over 210 triples. The test set has 209 unseen entities, distributed over 210 triples.

Parameters

- **clean_unseen** (*bool*) – If `True`, filters triples in validation and test sets that include entities not present in the training set.
- **check_md5hash** (*bool*) – If `True` check the md5hash of the datset files. Defaults to `False`.
- **add_reciprocal_rels** (*bool*) – Flag which specifies whether to add reciprocal relations. For every <s, p, o> in the dataset this creates a corresponding triple with reciprocal relation <o, p_reciprocal, s>. (default: False).

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)
```

load_yago3_10

ampligraph.datasets.**load_yago3_10**(*check_md5hash=False*, *clean_unseen=True*, *add_reciprocal_rels=False*)

Load the YAGO3-10 dataset

The dataset is a split of YAGO3 [MBS13], and has been first presented in [DMSR18].

The YAGO3-10 dataset is loaded from file if it exists at the AMPLIGRAPH_DATA_HOME location. If AMPLIGRAPH_DATA_HOME is not set the the default ~/ampligraph_datasets is checked.

If the dataset is not found at either location it is downloaded and placed in AMPLIGRAPH_DATA_HOME or ~/ampligraph_datasets.

It is divided in three splits:

- train
- valid
- test

Dataset	Train	Valid	Test	Entities	Relations
YAGO3-10	1,079,040	5,000	5,000	123,182	37

Parameters

- **check_md5hash** (*boolean*) – If True check the md5hash of the files. Defaults to False.
- **clean_unseen** (*bool*) – If True, filters triples in validation and test sets that include entities not present in the training set.
- **add_reciprocal_rels** (*bool*) – Flag which specifies whether to add reciprocal relations. For every <s, p, o> in the dataset this creates a corresponding triple with reciprocal relation <o, p_reciprocal, s>. (default: False).

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_fb15k

ampligraph.datasets.**load_fb15k**(*check_md5hash=False*, *add_reciprocal_rels=False*)
Load the FB15k dataset

> **Warning:** The dataset includes a large number of inverse relations that spilled to the test set, and its use in experiments has been deprecated. Use FB15k-237 instead.

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

The FB15k dataset is loaded from file if it exists at the `AMPLIGRAPH_DATA_HOME` location. If `AMPLIGRAPH_DATA_HOME` is not set the the default `~/ampligraph_datasets` is checked.

If the dataset is not found at either location it is downloaded and placed in `AMPLIGRAPH_DATA_HOME` or `~/ampligraph_datasets`.

The dataset is divided in three splits:

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

Dataset	Train	Valid	Test	Entities	Relations
FB15K	483,142	50,000	59,071	14,951	1,345

Parameters

- **check_md5hash** (*boolean*) – If `True` check the md5hash of the files. Defaults to `False`.
- **add_reciprocal_rels** (*bool*) – Flag which specifies whether to add reciprocal relations. For every <s, p, o> in the dataset this creates a corresponding triple with reciprocal relation <o, p_reciprocal, s>. (default: False).

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_wn18

ampligraph.datasets.**load_wn18**(*check_md5hash=False*, *add_reciprocal_rels=False*)
Load the WN18 dataset

> **Warning:** The dataset includes a large number of inverse relations that spilled to the test set, and its use in experiments has been deprecated. Use WN18RR instead.

WN18 is a subset of Wordnet. It was first presented by [BUGD+13].

The WN18 dataset is loaded from file if it exists at the `AMPLIGRAPH_DATA_HOME` location. If `AMPLIGRAPH_DATA_HOME` is not set the the default `~/ampligraph_datasets` is checked.

If the dataset is not found at either location it is downloaded and placed in `AMPLIGRAPH_DATA_HOME` or `~/ampligraph_datasets`.

The dataset is divided in three splits:

- `train`: 141,442 triples
- `valid` 5,000 triples
- `test` 5,000 triples

Dataset	Train	Valid	Test	Entities	Relations
WN18	141,442	5,000	5,000	40,943	18

Parameters

- **check_md5hash** (*bool*) – If `True` check the md5hash of the files. Defaults to `False`.
- **add_reciprocal_rels** (*bool*) – Flag which specifies whether to add reciprocal relations. For every <s, p, o> in the dataset this creates a corresponding triple with reciprocal relation <o, p_reciprocal, s>. (default: False).

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_wn11

ampligraph.datasets.**load_wn11**(*check_md5hash=False*, *clean_unseen=True*, *add_reciprocal_rels=False*)

Load the WordNet11 (WN11) dataset

WordNet was originally proposed in *WordNet: a lexical database for English* [Mil95].

WN11 dataset is loaded from file if it exists at the `AMPLIGRAPH_DATA_HOME` location. If `AMPLIGRAPH_DATA_HOME` is not set the the default `~/ampligraph_datasets` is checked.

If the dataset is not found at either location, it is downloaded and placed in `AMPLIGRAPH_DATA_HOME` or `~/ampligraph_datasets`.

It is divided in three splits:

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

Both the validation and test splits are associated with labels (binary ndarrays), with *True* for positive statements and *False* for negatives:

- `valid_labels`
- `test_labels`

Dataset	Train	Valid Pos	Valid Neg	Test Pos	Test Neg	Entities	Relations
WN11	110361	2606	2609	10493	10542	38588	11

Parameters

- **check_md5hash** (*boolean*) – If `True` check the md5hash of the files. Defaults to `False`.
- **clean_unseen** (*bool*) – If `True`, filters triples in validation and test sets that include entities not present in the training set.
- **add_reciprocal_rels** (*bool*) – Flag which specifies whether to add reciprocal relations. For every <s, p, o> in the dataset this creates a corresponding triple with reciprocal relation <o, p_reciprocal, s>. (default: False).

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

Return type dict

Examples

```
>>> from ampligraph.datasets import load_wn11
>>> X = load_wn11()
>>> X["valid"][0]
array(['__genus_xylomelum_1', '_type_of', '__dicot_genus_1'], dtype=object)
>>> X["valid_labels"][0:3]
array([ True, False,  True])
```

load_fb13

ampligraph.datasets.**load_fb13**(*check_md5hash=False*, *clean_unseen=True*, *add_reciprocal_rels=False*)

Load the Freebase13 (FB13) dataset

FB13 is a subset of Freebase [BEP+08] and was initially presented in *Reasoning With Neural Tensor Networks for Knowledge Base Completion* [SCMN13].

FB13 dataset is loaded from file if it exists at the `AMPLIGRAPH_DATA_HOME` location. If `AMPLIGRAPH_DATA_HOME` is not set the the default `~/ampligraph_datasets` is checked.

If the dataset is not found at either location, it is downloaded and placed in `AMPLIGRAPH_DATA_HOME` or `~/ampligraph_datasets`.

It is divided in three splits:

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

Both the validation and test splits are associated with labels (binary ndarrays), with *True* for positive statements and *False* for negatives:

- `valid_labels`
- `test_labels`

Dataset	Train	Valid Pos	Valid Neg	Test Pos	Test Neg	Entities	Relations
FB13	316232	5908	5908	23733	23731	75043	13

Parameters

- **check_md5hash** (*boolean*) – If `True` check the md5hash of the files. Defaults to `False`.
- **clean_unseen** (*bool*) – If `True`, filters triples in validation and test sets that include entities not present in the training set.
- **add_reciprocal_rels** (*bool*) – Flag which specifies whether to add reciprocal relations. For every <s, p, o> in the dataset this creates a corresponding triple with reciprocal relation <o, p_reciprocal, s>. (default: False).

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

Return type dict

Examples

```
>>> from ampligraph.datasets import load_fb13
>>> X = load_fb13()
>>> X["valid"][0]
array(['cornelie_van_zanten', 'gender', 'female'], dtype=object)
>>> X["valid_labels"][0:3]
array([True, False, True], dtype=object)
```

Datasets 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
WN11	110,361	5,215	21,035	38,194	11
FB13	316,232	11,816	47,464	75,043	13

Warning: WN18 and FB15k include a large number of inverse relations, and its use in experiments has been deprecated. **Use WN18RR and FB15K-237 instead**.

Warning: FB15K-237's validation set contains 8 unseen entities over 9 triples. The test set has 29 unseen entities, distributed over 28 triples. WN18RR's validation set contains 198 unseen entities over 210 triples. The test set has 209 unseen entities, distributed over 210 triples.

Note: WN11 and FB13 also provide true and negative labels for the triples in the validation and tests sets. In both cases the positive base rate is close to 50%.

Loaders for Custom Knowledge Graphs

Functions to load custom knowledge graphs from disk.

load_from_csv(directory_path, file_name[, ...])	Load a knowledge graph from a csv file
load_from_ntriples(folder_name, file_name[, ...])	Load RDF ntriples
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*, *add_reciprocal_rels=False*)

Load a knowledge graph from a csv file

Loads a knowledge graph serialized in a csv file as:

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

Note: The function filters duplicated statements.

Note: It is recommended to use *ampligraph.evaluation.train_test_split_no_unseen()* to split custom knowledge graphs into train, validation, and test sets. Using this function will lead to validation, test sets that do not include triples with entities that do not occur in the training set.

Parameters

- **directory_path** (*str*) – Folder where the input 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.
- **add_reciprocal_rels** (*bool*) – Flag which specifies whether to add reciprocal relations. For every <s, p, o> in the dataset this creates a corresponding triple with reciprocal relation <o, p_reciprocal, s>. (default: False)

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*, *add_reciprocal_rels=False*)

Load RDF ntriples

Loads an RDF knowledge graph serialized as ntriples, without building an RDF graph in memory. This function should be preferred over `load_from_rdf()`, since it does not load the graph into an rdflib model (and it is therefore faster by order of magnitudes). Nevertheless, it requires a ntriples serialization as in the example below:

```
_:alice <http://xmlns.com/foaf/0.1/knows> _:bob .
_:bob <http://xmlns.com/foaf/0.1/knows> _:alice .
```

Note: It is recommended to use *ampligraph.evaluation.train_test_split_no_unseen()* to split custom knowledge graphs into train, validation, and test sets. Using this function will lead to validation, test sets that do not include triples with entities that do not occur in the training set.

Parameters

- **folder_name** (*str*) – base folder where the file is stored.
- **file_name** (*str*) – file name
- **data_home** (*str*) – The path to the folder that contains the datasets.
- **add_reciprocal_rels** (*bool*) – Flag which specifies whether to add reciprocal relations. For every <s, p, o> in the dataset this creates a corresponding triple with reciprocal relation <o, p_reciprocal, s>. (default: False).

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*, *rdf_format='nt'*, *data_home=None*, *add_reciprocal_rels=False*)

Load an RDF file

Loads an RDF knowledge graph using rdflib APIs. Multiple RDF serialization formats are supported (nt, ttl, rdf/xml, etc). The entire graph will be loaded in memory, and converted into an rdflib *Graph* object.

Warning: Large RDF graphs should be serialized to ntriples beforehand and loaded with `load_from_ntriples()` instead.

Note: It is recommended to use *ampligraph.evaluation.train_test_split_no_unseen()* to split custom knowledge graphs into train, validation, and test sets. Using this function will lead to validation, test sets that do not include triples with entities that do not occur in the training set.

Parameters

- **folder_name** (*str*) – Base folder where the file is stored.

- **file_name** (*str*) – File name.
- **rdf_format** (*str*) – The RDF serialization format (nt, ttl, rdf/xml - see rdflib documentation).
- **data_home** (*str*) – The path to the folder that contains the datasets.
- **add_reciprocal_rels** (*bool*) – Flag which specifies whether to add reciprocal relations. For every <s, p, o> in the dataset this creates a corresponding triple with reciprocal relation <o, p_reciprocal, s>. (default: False).

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

Return type ndarray , shape [n, 3]

Hint: AmpliGraph includes a helper function to split a generic knowledge graphs into **training**, **validation**, and **test** sets. See *ampligraph.evaluation.train_test_split_no_unseen()*.

3.3.2 Models

Knowledge Graph Embedding Models

RandomBaseline([seed, verbose])	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
ConvE([k, eta, epochs, batches_count, seed, ...])	Convolutional 2D KG Embeddings
ConvKB([k, eta, epochs, batches_count, ...])	Convolution-based model

RandomBaseline

class ampligraph.latent_features.**RandomBaseline**(*seed=0*, *verbose=False*)

Random baseline

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

The 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 actual 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, verbose])	Initialize the model
fit(X[, early_stopping, early_stopping_params])	Train the random model.
predict(X[, from_idx])	Predict the scores of triples using a trained embedding model.
get_hyperparameter_dict()	Returns hyperparameters of the model.

__init__ (*seed=0*, *verbose=False*)
Initialize the model

Parameters

- **seed** (*int*) – The seed used by the internal random numbers generator.
- **verbose** (*bool*) – Verbose mode.

fit (*X*, *early_stopping=False*, *early_stopping_params={}*)
Train the random model.

There is no actual training involved in practice and the early stopping parameters won't have any effect.

Parameters

- **X** (*ndarray, shape [n, 3]*) – The training triples
- **early_stopping** (*bool*) – Flag to enable early stopping (default:False).

  If set to `True`, the training loop adopts the following early stopping heuristic:

  – The model will be trained regardless of early stopping for `burn_in` epochs.

  – Every `check_interval` epochs the method will compute the metric specified in `criteria`.

  If such metric decreases for `stop_interval` checks, we stop training early.

  Note the metric is computed on `x_valid`. This is usually a validation set that you held out.

  Also, because `criteria` is a ranking metric, it requires generating negatives. Entities used to generate corruptions can be specified, as long as the side(s) of a triple to corrupt.

The method supports filtered metrics, by passing an array of positives to `x_filter`. This will be used to filter the negatives generated on the fly (i.e. the corruptions).

Note: Keep in mind the early stopping criteria may introduce a certain overhead (caused by the metric computation). The goal is to strike a good trade-off between such overhead and saving training epochs.

A common approach is to use MRR unfiltered:

```
early_stopping_params={x_valid=X['valid'], 'criteria': 'mrr'}
```

Note the size of validation set also contributes to such overhead. In most cases a smaller validation set would be enough.

- **early_stopping_params** (*dictionary*) – Dictionary of hyperparameters for the early stopping heuristics.

  The following string 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] : Positive triples to use as filter if a 'filtered' early stopping criteria is desired (i.e. filtered-MRR if 'criteria':'mrr'). Note this will affect training time (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)
  - **'corruption_entities'**: List of entities to be used for corruptions. If 'all', it uses all entities (default: 'all')
  - **'corrupt_side'**: Specifies which side to corrupt. 's', 'o', 's+o' (default)

  Example: `early_stopping_params={x_valid=X['valid'], 'criteria': 'mrr'}`

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

Predict the scores of triples using a trained embedding model. The function returns raw scores generated by the model.

Note: To obtain probability estimates, calibrate the model with *calibrate()*, then call *predict_proba()*.

Parameters

- **X** (*ndarray, shape [n, 3]*) – The triples to score.
- **from_idx** (*bool*) – If True, will skip conversion to internal IDs. (default: False).

Returns **scores_predict** – The predicted scores for input triples X.

Return type ndarray, shape [n]

get_hyperparameter_dict()
Returns hyperparameters of the model.

Returns **hyperparam_dict** – Dictionary of hyperparameters that were used for training.

Return type dict

TransE

class ampligraph.latent_features.**TransE**(*k=100, eta=2, epochs=100, batches_count=100, seed=0, embedding_model_params={'corrupt_sides': ['s,o'], 'negative_corruption_entities': 'all', 'norm': 1, 'normalize_ent_emb': False}, optimizer='adam', optimizer_params={'lr': 0.0005}, loss='nll', loss_params={}, regularizer=None, regularizer_params={}, initializer='xavier', initializer_params={'uniform': False}, verbose=False*)

Translating Embeddings (TransE)

The model as described in [BUGD+13].

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.

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']]))
[-4.6903257, -3.9047198]
>>> model.get_embeddings(['f','e'], embedding_type='entity')
array([[ 0.10673896, -0.28916815,  0.6278883 , -0.1194713 , -0.10372276,
-0.37258488,  0.06460134, -0.27879423,  0.25456288,  0.18665907],
[-0.64494324, -0.12939683,  0.3181001 ,  0.16745451, -0.03766293,
 0.24314676, -0.23038973, -0.658638  ,  0.5680542 , -0.05401703]],
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[, embedding_type])	Get the embeddings of entities or relations.
get_hyperparameter_dict()	Returns hyperparameters of the model.
predict(X[, from_idx])	Predict the scores of triples using a trained embedding model.
calibrate(X_pos[, X_neg, ...])	Calibrate predictions
predict_proba(X)	Predicts probabilities using the Platt scaling model (after calibration).

__init__(*k=100, eta=2, epochs=100, batches_count=100, seed=0, embedding_model_params={'corrupt_sides': ['s,o'], 'negative_corruption_entities': 'all', 'norm': 1, 'normalize_ent_emb': False}, optimizer='adam', optimizer_params={'lr': 0.0005}, loss='nll', loss_params={}, regularizer=None, regularizer_params={}, initializer='xavier', initializer_params={'uniform': False}, verbose=False*)

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, passed to the model as a dictionary.

  Supported keys:

  - **'norm'** (int): the norm to be used in the scoring function (1 or 2-norm - default: 1).
  - **'normalize_ent_emb'** (bool): flag to indicate whether to normalize entity embeddings after each batch update (default: False).
  - **negative_corruption_entities** : entities to be used for generation of corruptions while training. It can take the following values : `all` (default: all entities), `batch` (entities present in each batch), list of entities or an int (which indicates how many entities that should be used for corruption generation).
  - **corrupt_sides** : Specifies how to generate corruptions for training. Takes values *s*, *o*, *s+o* or any combination passed as a list.

  Example: `embedding_model_params={'norm': 1, 'normalize_ent_emb': False}`
- **optimizer** (*string*) – The optimizer used to minimize the loss function. Choose between 'sgd', 'adagrad', 'adam', 'momentum'.
- **optimizer_params** (*dict*) – Arguments specific to the optimizer, passed as a dictionary.

Supported keys:

- **'lr'** (float): learning rate (used by all the optimizers). Default: 0.1.
- **'momentum'** (float): learning momentum (only used when `optimizer=momentum`). Default: 0.9.

Example: `optimizer_params={'lr': 0.01}`

- **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.
  - `multiclass_nll` the model will use multiclass nll loss. Switch to multiclass loss defined in [aC15] by passing 'corrupt_sides' as ['s','o'] to embedding_model_params. To use loss defined in [KBK17] pass 'corrupt_sides' as 'o' to embedding_model_params.
- **loss_params** (*dict*) – Dictionary of loss-specific hyperparameters. See *loss functions* documentation for additional details.

  Example: `optimizer_params={'lr': 0.01}` if `loss='pairwise'`.
- **regularizer** (*string*) – The regularization strategy to use with the loss function.
  - `None`: the model will not use any regularizer (default)
  - 'LP': the model will use L1, L2 or L3 based on the value of `regularizer_params['p']` (see below).
- **regularizer_params** (*dict*) – Dictionary of regularizer-specific hyperparameters. See the *regularizers* documentation for additional details.

  Example: `regularizer_params={'lambda': 1e-5, 'p': 2}` if `regularizer='LP'`.
- **initializer** (*string*) – The type of initializer to use.
  - `normal`: The embeddings will be initialized from a normal distribution
  - `uniform`: The embeddings will be initialized from a uniform distribution
  - `xavier`: The embeddings will be initialized using xavier strategy (default)
- **initializer_params** (*dict*) – Dictionary of initializer-specific hyperparameters. See the *initializer* documentation for additional details.

  Example: `initializer_params={'mean': 0, 'std': 0.001}` if `initializer='normal'`.

verbose [bool] Verbose mode

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).

  If set to `True`, the training loop adopts the following early stopping heuristic:

  – The model will be trained regardless of early stopping for `burn_in` epochs.

  – Every `check_interval` epochs the method will compute the metric specified in `criteria`.

  If such metric decreases for `stop_interval` checks, we stop training early.

  Note the metric is computed on `x_valid`. This is usually a validation set that you held out.

  Also, because `criteria` is a ranking metric, it requires generating negatives. Entities used to generate corruptions can be specified, as long as the side(s) of a triple to corrupt. The method supports filtered metrics, by passing an array of positives to `x_filter`. This will be used to filter the negatives generated on the fly (i.e. the corruptions).

  Note: Keep in mind the early stopping criteria may introduce a certain overhead (caused by the metric computation). The goal is to strike a good trade-off between such overhead and saving training epochs.

  A common approach is to use MRR unfiltered:

  ```
  early_stopping_params={x_valid=X['valid'], 'criteria': 'mrr'}
  ```

  Note the size of validation set also contributes to such overhead. In most cases a smaller validation set would be enough.

- **early_stopping_params** (*dictionary*) – Dictionary of hyperparameters for the early stopping heuristics.

  The following string 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] : Positive triples to use as filter if a 'filtered' early stopping criteria is desired (i.e. filtered-MRR if 'criteria':'mrr'). Note this will affect training time (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)

  – **'corruption_entities'**: List of entities to be used for corruptions. If 'all', it uses all entities (default: 'all')

  – **'corrupt_side'**: Specifies which side to corrupt. 's', 'o', 's+o' (default)

  Example: `early_stopping_params={x_valid=X['valid'], 'criteria': 'mrr'}`

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

Note: Use *ampligraph.utils.create_tensorboard_visualizations()* to visualize the embeddings with TensorBoard.

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.
- **embedding_type** (*string*) – If 'entity', `entities` argument will be considered as a list of knowledge graph entities (i.e. nodes). If set to 'relation', they will be treated as relation types instead (i.e. predicates).

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

Return type ndarray, shape [n, k]

get_hyperparameter_dict()
Returns hyperparameters of the model.

Returns **hyperparam_dict** – Dictionary of hyperparameters that were used for training.

Return type dict

predict(*X*, *from_idx=False*)
Predict the scores of triples using a trained embedding model. The function returns raw scores generated by the model.

Note: To obtain probability estimates, calibrate the model with *calibrate()*, then call *predict_proba()*.

Parameters

- **X** (*ndarray, shape [n, 3]*) – The triples to score.
- **from_idx** (*bool*) – If True, will skip conversion to internal IDs. (default: False).

Returns **scores_predict** – The predicted scores for input triples X.

Return type ndarray, shape [n]

calibrate(*X_pos*, *X_neg=None*, *positive_base_rate=None*, *batches_count=100*, *epochs=50*)
Calibrate predictions

The method implements the heuristics described in [TC20], using Platt scaling [P+99].

The calibrated predictions can be obtained with *predict_proba()* after calibration is done.

Ideally, calibration should be performed on a validation set that was not used to train the embeddings.

There are two modes of operation, depending on the availability of negative triples:

1. Both positive and negative triples are provided via `X_pos` and `X_neg` respectively. The optimization is done using a second-order method (limited-memory BFGS), therefore no hyperparameter needs to be specified.

2. Only positive triples are provided, and the negative triples are generated by corruptions just like it is done in training or evaluation. The optimization is done using a first-order method (ADAM), therefore `batches_count` and `epochs` must be specified.

Calibration is highly dependent on the base rate of positive triples. Therefore, for mode (2) of operation, the user is required to provide the `positive_base_rate` argument. For mode (1), that can be inferred automatically by the relative sizes of the positive and negative sets, but the user can override that by providing a value to `positive_base_rate`.

Defining the positive base rate is the biggest challenge when calibrating without negatives. That depends on the user choice of which triples will be evaluated during test time. Let's take WN11 as an example: it has around 50% positives triples on both the validation set and test set, so naturally the positive base rate is 50%. However, should the user resample it to have 75% positives and 25% negatives, its previous calibration will be degraded. The user must recalibrate the model now with a 75% positive base rate. Therefore, this parameter depends on how the user handles the dataset and cannot be determined automatically or a priori.

Note: Incompatible with large graph mode (i.e. if `self.dealing_with_large_graphs=True`).

Note: [TC20] calibration experiments available here.

Parameters

- **X_pos** (*ndarray (shape [n, 3])*) – Numpy array of positive triples.
- **X_neg** (*ndarray (shape [n, 3])*) – Numpy array of negative triples.

  If *None*, the negative triples are generated via corruptions and the user must provide a positive base rate instead.
- **positive_base_rate** (*float*) – Base rate of positive statements.

  For example, if we assume there is a fifty-fifty chance of any query to be true, the base rate would be 50%.

  If `X_neg` is provided and this is *None*, the relative sizes of `X_pos` and `X_neg` will be used to determine the base rate. For example, if we have 50 positive triples and 200 negative triples, the positive base rate will be assumed to be 50/(50+200) = 1/5 = 0.2.

  This must be a value between 0 and 1.
- **batches_count** (*int*) – Number of batches to complete one epoch of the Platt scaling training. Only applies when `X_neg` is *None*.
- **epochs** (*int*) – Number of epochs used to train the Platt scaling model. Only applies when `X_neg` is *None*.

Examples

```
>>> import numpy as np
>>> from sklearn.metrics import brier_score_loss, log_loss
>>> from scipy.special import expit
>>>
>>> from ampligraph.datasets import load_wn11
>>> from ampligraph.latent_features.models import TransE
>>>
>>> X = load_wn11()
>>> X_valid_pos = X['valid'][X['valid_labels']]
>>> X_valid_neg = X['valid'][~X['valid_labels']]
>>>
>>> model = TransE(batches_count=64, seed=0, epochs=500, k=100, eta=20,
>>>                optimizer='adam', optimizer_params={'lr':0.0001},
>>>                loss='pairwise', verbose=True)
>>>
>>> model.fit(X['train'])
>>>
>>> # Raw scores
>>> scores = model.predict(X['test'])
>>>
>>> # Calibrate with positives and negatives
>>> model.calibrate(X_valid_pos, X_valid_neg, positive_base_rate=None)
>>> probas_pos_neg = model.predict_proba(X['test'])
>>>
>>> # Calibrate with just positives and base rate of 50%
>>> model.calibrate(X_valid_pos, positive_base_rate=0.5)
>>> probas_pos = model.predict_proba(X['test'])
>>>
>>> # Calibration evaluation with the Brier score loss (the smaller, the␣
↪better)
>>> print("Brier scores")
>>> print("Raw scores:", brier_score_loss(X['test_labels'], expit(scores)))
>>> print("Positive and negative calibration:", brier_score_loss(X['test_
↪labels'], probas_pos_neg))
>>> print("Positive only calibration:", brier_score_loss(X['test_labels'],␣
↪probas_pos))
Brier scores
Raw scores: 0.4925058891371126
Positive and negative calibration: 0.20434617882733366
Positive only calibration: 0.22597599585144656
```

predict_proba(*X*)

Predicts probabilities using the Platt scaling model (after calibration).

Model must be calibrated beforehand with the `calibrate` method.

Parameters **X** (*ndarray (shape [n, 3])*) – Numpy array of triples to be evaluated.

Returns **probas** – Probability of each triple to be true according to the Platt scaling calibration.

Return type ndarray (shape [n])

DistMult

class ampligraph.latent_features.**DistMult**(*k=100, eta=2, epochs=100, batches_count=100, seed=0, embedding_model_params={'corrupt_sides': ['s,o'], 'negative_corruption_entities': 'all', 'normalize_ent_emb': False}, optimizer='adam', optimizer_params={'lr': 0.0005}, loss='nll', loss_params={}, regularizer=None, regularizer_params={}, initializer='xavier', initializer_params={'uniform': False}, verbose=False*)

The DistMult model

The model as described in [YYH+14].

The bilinear diagonal DistMult model uses the trilinear dot product as scoring function:

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

where $\mathbf{e}_s$ is the embedding of the subject, $\mathbf{r}_p$ the embedding of the predicate and $\mathbf{e}_o$ the embedding of the object.

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']]))
[-0.13863425, -0.09917116]
>>> model.get_embeddings(['f','e'], embedding_type='entity')
array([[ 0.10137264, -0.28248304,  0.6153027 , -0.13133956, -0.11675504,
-0.37876177,  0.06027773, -0.26390398,  0.254603  ,  0.1888549 ],
[-0.6467299 , -0.13729756,  0.3074872 ,  0.16966867, -0.04098966,
 0.25289047, -0.2212451 , -0.6527815 ,  0.5657673 , -0.03876532]],
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[, embedding_type])	Get the embeddings of entities or relations.
get_hyperparameter_dict()	Returns hyperparameters of the model.
predict(X[, from_idx])	Predict the scores of triples using a trained embedding model.
calibrate(X_pos[, X_neg, ...])	Calibrate predictions
predict_proba(X)	Predicts probabilities using the Platt scaling model (after calibration).

__init__(*k=100, eta=2, epochs=100, batches_count=100, seed=0, embedding_model_params={'corrupt_sides': ['s,o'], 'negative_corruption_entities': 'all', 'normalize_ent_emb': False}, optimizer='adam', optimizer_params={'lr': 0.0005}, loss='nll', loss_params={}, regularizer=None, regularizer_params={}, initializer='xavier', initializer_params={'uniform': False}, verbose=False*)

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, passed to the model as a dictionary.

  Supported keys:

  - **'normalize_ent_emb'** (bool): flag to indicate whether to normalize entity embeddings after each batch update (default: False).
  - **'negative_corruption_entities'** - Entities to be used for generation of corruptions while training. It can take the following values : `all` (default: all entities), `batch` (entities present in each batch), list of entities or an int (which indicates how many entities that should be used for corruption generation).
  - **corrupt_sides** : Specifies how to generate corruptions for training. Takes values *s*, *o*, *s+o* or any combination passed as a list

  Example: `embedding_model_params={'normalize_ent_emb': False}`
- **optimizer** (*string*) – The optimizer used to minimize the loss function. Choose between 'sgd', 'adagrad', 'adam', 'momentum'.
- **optimizer_params** (*dict*) – Arguments specific to the optimizer, passed as a dictionary.

  Supported keys:

- **'lr'** (float): learning rate (used by all the optimizers). Default: 0.1.
- **'momentum'** (float): learning momentum (only used when `optimizer=momentum`). Default: 0.9.

Example: `optimizer_params={'lr': 0.01}`

- **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.
  - `multiclass_nll` the model will use multiclass nll loss. Switch to multiclass loss defined in [aC15] by passing 'corrupt_sides' as ['s','o'] to embedding_model_params. To use loss defined in [KBK17] pass 'corrupt_sides' as 'o' to embedding_model_params.
- **loss_params** (*dict*) – Dictionary of loss-specific hyperparameters. See *loss functions* documentation for additional details.

  Example: `optimizer_params={'lr': 0.01}` if `loss='pairwise'`.
- **regularizer** (*string*) – The regularization strategy to use with the loss function.
  - `None`: the model will not use any regularizer (default)
  - 'LP': the model will use L1, L2 or L3 based on the value of `regularizer_params['p']` (see below).
- **regularizer_params** (*dict*) – Dictionary of regularizer-specific hyperparameters. See the *regularizers* documentation for additional details.

  Example: `regularizer_params={'lambda': 1e-5, 'p': 2}` if `regularizer='LP'`.
- **initializer** (*string*) – The type of initializer to use.
  - `normal`: The embeddings will be initialized from a normal distribution
  - `uniform`: The embeddings will be initialized from a uniform distribution
  - `xavier`: The embeddings will be initialized using xavier strategy (default)
- **initializer_params** (*dict*) – Dictionary of initializer-specific hyperparameters. See the *initializer* documentation for additional details.

  Example: `initializer_params={'mean': 0, 'std': 0.001}` if `initializer='normal'`.
- **verbose** (*bool*) – Verbose mode.

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).

  If set to `True`, the training loop adopts the following early stopping heuristic:

  – The model will be trained regardless of early stopping for `burn_in` epochs.

  – Every `check_interval` epochs the method will compute the metric specified in `criteria`.

  If such metric decreases for `stop_interval` checks, we stop training early.

  Note the metric is computed on `x_valid`. This is usually a validation set that you held out.

  Also, because `criteria` is a ranking metric, it requires generating negatives. Entities used to generate corruptions can be specified, as long as the side(s) of a triple to corrupt. The method supports filtered metrics, by passing an array of positives to `x_filter`. This will be used to filter the negatives generated on the fly (i.e. the corruptions).

  Note: Keep in mind the early stopping criteria may introduce a certain overhead (caused by the metric computation). The goal is to strike a good trade-off between such overhead and saving training epochs.

  A common approach is to use MRR unfiltered:

  ```
  early_stopping_params={x_valid=X['valid'], 'criteria': 'mrr'}
  ```

  Note the size of validation set also contributes to such overhead. In most cases a smaller validation set would be enough.

- **early_stopping_params** (*dictionary*) – Dictionary of hyperparameters for the early stopping heuristics.

  The following string 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] : Positive triples to use as filter if a 'filtered' early stopping criteria is desired (i.e. filtered-MRR if 'criteria':'mrr'). Note this will affect training time (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)

  – **'corruption_entities'**: List of entities to be used for corruptions. If 'all', it uses all entities (default: 'all')

  – **'corrupt_side'**: Specifies which side to corrupt. 's', 'o', 's+o' (default)

  Example: `early_stopping_params={x_valid=X['valid'], 'criteria': 'mrr'}`

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

Note: Use *ampligraph.utils.create_tensorboard_visualizations()* to visualize the embeddings with TensorBoard.

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.
- **embedding_type** (*string*) – If 'entity', `entities` argument will be considered as a list of knowledge graph entities (i.e. nodes). If set to 'relation', they will be treated as relation types instead (i.e. predicates).

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

Return type ndarray, shape [n, k]

get_hyperparameter_dict()
Returns hyperparameters of the model.

Returns **hyperparam_dict** – Dictionary of hyperparameters that were used for training.

Return type dict

predict(*X*, *from_idx=False*)
Predict the scores of triples using a trained embedding model. The function returns raw scores generated by the model.

Note: To obtain probability estimates, calibrate the model with *calibrate()*, then call *predict_proba()*.

Parameters

- **X** (*ndarray, shape [n, 3]*) – The triples to score.
- **from_idx** (*bool*) – If True, will skip conversion to internal IDs. (default: False).

Returns **scores_predict** – The predicted scores for input triples X.

Return type ndarray, shape [n]

calibrate(*X_pos*, *X_neg=None*, *positive_base_rate=None*, *batches_count=100*, *epochs=50*)
Calibrate predictions

The method implements the heuristics described in [TC20], using Platt scaling [P+99].

The calibrated predictions can be obtained with *predict_proba()* after calibration is done.

Ideally, calibration should be performed on a validation set that was not used to train the embeddings.

There are two modes of operation, depending on the availability of negative triples:

1. Both positive and negative triples are provided via `X_pos` and `X_neg` respectively. The optimization is done using a second-order method (limited-memory BFGS), therefore no hyperparameter needs to be specified.

2. Only positive triples are provided, and the negative triples are generated by corruptions just like it is done in training or evaluation. The optimization is done using a first-order method (ADAM), therefore `batches_count` and `epochs` must be specified.

Calibration is highly dependent on the base rate of positive triples. Therefore, for mode (2) of operation, the user is required to provide the `positive_base_rate` argument. For mode (1), that can be inferred automatically by the relative sizes of the positive and negative sets, but the user can override that by providing a value to `positive_base_rate`.

Defining the positive base rate is the biggest challenge when calibrating without negatives. That depends on the user choice of which triples will be evaluated during test time. Let's take WN11 as an example: it has around 50% positives triples on both the validation set and test set, so naturally the positive base rate is 50%. However, should the user resample it to have 75% positives and 25% negatives, its previous calibration will be degraded. The user must recalibrate the model now with a 75% positive base rate. Therefore, this parameter depends on how the user handles the dataset and cannot be determined automatically or a priori.

Note: Incompatible with large graph mode (i.e. if `self.dealing_with_large_graphs=True`).

Note: [TC20] calibration experiments available here.

Parameters

- **X_pos** (*ndarray (shape [n, 3])*) – Numpy array of positive triples.
- **X_neg** (*ndarray (shape [n, 3])*) – Numpy array of negative triples.

  If *None*, the negative triples are generated via corruptions and the user must provide a positive base rate instead.
- **positive_base_rate** (*float*) – Base rate of positive statements.

  For example, if we assume there is a fifty-fifty chance of any query to be true, the base rate would be 50%.

  If `X_neg` is provided and this is *None*, the relative sizes of `X_pos` and `X_neg` will be used to determine the base rate. For example, if we have 50 positive triples and 200 negative triples, the positive base rate will be assumed to be 50/(50+200) = 1/5 = 0.2.

  This must be a value between 0 and 1.
- **batches_count** (*int*) – Number of batches to complete one epoch of the Platt scaling training. Only applies when `X_neg` is *None*.
- **epochs** (*int*) – Number of epochs used to train the Platt scaling model. Only applies when `X_neg` is *None*.

Examples

```
>>> import numpy as np
>>> from sklearn.metrics import brier_score_loss, log_loss
>>> from scipy.special import expit
>>>
>>> from ampligraph.datasets import load_wn11
>>> from ampligraph.latent_features.models import TransE
>>>
>>> X = load_wn11()
>>> X_valid_pos = X['valid'][X['valid_labels']]
>>> X_valid_neg = X['valid'][~X['valid_labels']]
>>>
>>> model = TransE(batches_count=64, seed=0, epochs=500, k=100, eta=20,
>>>                optimizer='adam', optimizer_params={'lr':0.0001},
>>>                loss='pairwise', verbose=True)
>>>
>>> model.fit(X['train'])
>>>
>>> # Raw scores
>>> scores = model.predict(X['test'])
>>>
>>> # Calibrate with positives and negatives
>>> model.calibrate(X_valid_pos, X_valid_neg, positive_base_rate=None)
>>> probas_pos_neg = model.predict_proba(X['test'])
>>>
>>> # Calibrate with just positives and base rate of 50%
>>> model.calibrate(X_valid_pos, positive_base_rate=0.5)
>>> probas_pos = model.predict_proba(X['test'])
>>>
>>> # Calibration evaluation with the Brier score loss (the smaller, the␣
↪better)
>>> print("Brier scores")
>>> print("Raw scores:", brier_score_loss(X['test_labels'], expit(scores)))
>>> print("Positive and negative calibration:", brier_score_loss(X['test_
↪labels'], probas_pos_neg))
>>> print("Positive only calibration:", brier_score_loss(X['test_labels'],␣
↪probas_pos))
Brier scores
Raw scores: 0.4925058891371126
Positive and negative calibration: 0.20434617882733366
Positive only calibration: 0.22597599585144656
```

predict_proba(*X*)

Predicts probabilities using the Platt scaling model (after calibration).

Model must be calibrated beforehand with the `calibrate` method.

Parameters **X** (*ndarray (shape [n, 3])*) – Numpy array of triples to be evaluated.

Returns **probas** – Probability of each triple to be true according to the Platt scaling calibration.

Return type ndarray (shape [n])

ComplEx

class ampligraph.latent_features.**ComplEx**(*k=100, eta=2, epochs=100, batches_count=100, seed=0, embedding_model_params={'corrupt_sides': ['s,o'], 'negative_corruption_entities': 'all'}, optimizer='adam', optimizer_params={'lr': 0.0005}, loss='nll', loss_params={}, regularizer=None, regularizer_params={}, initializer='xavier', initializer_params={'uniform': False}, verbose=False*)

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 $\mathcal{C}$:

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

ComplEx can be improved if used alongside the nuclear 3-norm (the **ComplEx-N3** model [LUO18]), which can be easily added to the loss function via the `regularizer` hyperparameter with `p=3` and a chosen regularisation weight (represented by `lambda`), as shown in the example below. See also *ampligraph.latent_features.LPRegularizer()*.

Note: Since ComplEx embeddings belong to $\mathcal{C}$, this model uses twice as many parameters as *ampligraph.latent_features.DistMult*.

Examples

```
>>> import numpy as np
>>> from ampligraph.latent_features import ComplEx
>>>
>>> model = ComplEx(batches_count=2, seed=555, epochs=100, k=20, eta=5,
>>>             loss='pairwise', loss_params={'margin':1},
>>>             regularizer='LP', regularizer_params={'p': 2, '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']]))
[[0.019520484], [-0.14998421]]
>>> model.get_embeddings(['f','e'], embedding_type='entity')
array([[-0.33021057,  0.26524785,  0.0446662 , -0.07932718, -0.15453218,
    -0.22342539, -0.03382565,  0.17444217,  0.03009969, -0.33569157,
     0.3200497 ,  0.03803705,  0.05536304, -0.00929996,  0.24446663,
     0.34408194,  0.16192885, -0.15033236, -0.19703785, -0.00783876,
     0.1495124 , -0.3578853 , -0.04975723, -0.03930473,  0.1663541 ,
    -0.24731971, -0.141296  ,  0.03150219,  0.15328223, -0.18549544,
    -0.39240393, -0.10824018,  0.03394471, -0.11075485,  0.1367736 ,
     0.10059565, -0.32808647, -0.00472086,  0.14231135, -0.13876757],
```

(continues on next page)

(continued from previous page)

```
  [-0.09483694,  0.3531292 ,  0.04992269, -0.07774793,  0.1635035 ,
    0.30610007,  0.3666711 , -0.13785957, -0.3143734 , -0.36909637,
   -0.13792469, -0.07069954, -0.0368113 , -0.16743314,  0.4090072 ,
   -0.03407392,  0.3113114 , -0.08418448,  0.21435146,  0.12006859,
    0.08447982, -0.02025972,  0.38752195,  0.11451488, -0.0258422 ,
   -0.10990044, -0.22661531, -0.00478273, -0.0238297 , -0.14207476,
    0.11064807,  0.20135397,  0.22501846, -0.1731076 , -0.2770435 ,
    0.30784574, -0.15043163, -0.11599299,  0.05718031, -0.1300622 ]],
dtype=float32)
```

Methods

__init__([k, eta, epochs, batches_count, ...])	Initialize an EmbeddingModel
fit(X[, early_stopping, early_stopping_params])	Train a ComplEx model.
get_embeddings(entities[, embedding_type])	Get the embeddings of entities or relations.
get_hyperparameter_dict()	Returns hyperparameters of the model.
predict(X[, from_idx])	Predict the scores of triples using a trained embedding model.
calibrate(X_pos[, X_neg, ...])	Calibrate predictions
predict_proba(X)	Predicts probabilities using the Platt scaling model (after calibration).

__init__(*k=100, eta=2, epochs=100, batches_count=100, seed=0, embedding_model_params={'corrupt_sides': ['s,o'], 'negative_corruption_entities': 'all'}, optimizer='adam', optimizer_params={'lr': 0.0005}, loss='nll', loss_params={}, regularizer=None, regularizer_params={}, initializer='xavier', initializer_params={'uniform': False}, verbose=False*)

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:
  - **'negative_corruption_entities'** - Entities to be used for generation of corruptions while training. It can take the following values : `all` (default: all entities), `batch` (entities present in each batch), list of entities or an int (which indicates how many entities that should be used for corruption generation).
  - **corrupt_sides** : Specifies how to generate corruptions for training. Takes values *s*, *o*, *s+o* or any combination passed as a list
- **optimizer** (*string*) – The optimizer used to minimize the loss function. Choose between 'sgd', 'adagrad', 'adam', 'momentum'.

- **optimizer_params** (*dict*) – Arguments specific to the optimizer, passed as a dictionary.

  Supported keys:

  - **'lr'** (float): learning rate (used by all the optimizers). Default: 0.1.
  - **'momentum'** (float): learning momentum (only used when `optimizer=momentum`). Default: 0.9.

  Example: `optimizer_params={'lr': 0.01}`

- **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.
  - `multiclass_nll` the model will use multiclass nll loss. Switch to multiclass loss defined in [aC15] by passing 'corrupt_sides' as ['s','o'] to embedding_model_params. To use loss defined in [KBK17] pass 'corrupt_sides' as 'o' to embedding_model_params.
- **loss_params** (*dict*) – Dictionary of loss-specific hyperparameters. See *loss functions* documentation for additional details.

  Example: `optimizer_params={'lr': 0.01}` if `loss='pairwise'`.
- **regularizer** (*string*) – The regularization strategy to use with the loss function.
  - `None`: the model will not use any regularizer (default)
  - 'LP': the model will use L1, L2 or L3 based on the value of `regularizer_params['p']` (see below).
- **regularizer_params** (*dict*) – Dictionary of regularizer-specific hyperparameters. See the *regularizers* documentation for additional details.

  Example: `regularizer_params={'lambda': 1e-5, 'p': 2}` if `regularizer='LP'`.
- **initializer** (*string*) – The type of initializer to use.
  - `normal`: The embeddings will be initialized from a normal distribution
  - `uniform`: The embeddings will be initialized from a uniform distribution
  - `xavier`: The embeddings will be initialized using xavier strategy (default)
- **initializer_params** (*dict*) – Dictionary of initializer-specific hyperparameters. See the *initializer* documentation for additional details.

  Example: `initializer_params={'mean': 0, 'std': 0.001}` if `initializer='normal'`.
- **verbose** (*bool*) – Verbose mode.

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).

  If set to `True`, the training loop adopts the following early stopping heuristic:

  – The model will be trained regardless of early stopping for `burn_in` epochs.

  – Every `check_interval` epochs the method will compute the metric specified in `criteria`.

  If such metric decreases for `stop_interval` checks, we stop training early.

  Note the metric is computed on `x_valid`. This is usually a validation set that you held out.

  Also, because `criteria` is a ranking metric, it requires generating negatives. Entities used to generate corruptions can be specified, as long as the side(s) of a triple to corrupt. The method supports filtered metrics, by passing an array of positives to `x_filter`. This will be used to filter the negatives generated on the fly (i.e. the corruptions).

  Note: Keep in mind the early stopping criteria may introduce a certain overhead (caused by the metric computation). The goal is to strike a good trade-off between such overhead and saving training epochs.

  A common approach is to use MRR unfiltered:

  ```
  early_stopping_params={x_valid=X['valid'], 'criteria': 'mrr'}
  ```

  Note the size of validation set also contributes to such overhead. In most cases a smaller validation set would be enough.

- **early_stopping_params** (*dictionary*) – Dictionary of hyperparameters for the early stopping heuristics.

  The following string 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] : Positive triples to use as filter if a 'filtered' early stopping criteria is desired (i.e. filtered-MRR if 'criteria':'mrr'). Note this will affect training time (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)

  – **'corruption_entities'**: List of entities to be used for corruptions. If 'all', it uses all entities (default: 'all')

  – **'corrupt_side'**: Specifies which side to corrupt. 's', 'o', 's+o' (default)

Example: `early_stopping_params={x_valid=X['valid'], 'criteria': 'mrr'}`

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

Note: Use *ampligraph.utils.create_tensorboard_visualizations()* to visualize the embeddings with TensorBoard.

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.
- **embedding_type** (*string*) – If 'entity', `entities` argument will be considered as a list of knowledge graph entities (i.e. nodes). If set to 'relation', they will be treated as relation types instead (i.e. predicates).

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

Return type ndarray, shape [n, k]

get_hyperparameter_dict()
Returns hyperparameters of the model.

Returns **hyperparam_dict** – Dictionary of hyperparameters that were used for training.

Return type dict

predict(*X*, *from_idx=False*)
Predict the scores of triples using a trained embedding model. The function returns raw scores generated by the model.

Note: To obtain probability estimates, calibrate the model with *calibrate()*, then call *predict_proba()*.

Parameters

- **X** (*ndarray, shape [n, 3]*) – The triples to score.
- **from_idx** (*bool*) – If True, will skip conversion to internal IDs. (default: False).

Returns **scores_predict** – The predicted scores for input triples X.

Return type ndarray, shape [n]

calibrate(*X_pos*, *X_neg=None*, *positive_base_rate=None*, *batches_count=100*, *epochs=50*)
Calibrate predictions

The method implements the heuristics described in [TC20], using Platt scaling [P+99].

The calibrated predictions can be obtained with *predict_proba()* after calibration is done.

Ideally, calibration should be performed on a validation set that was not used to train the embeddings.

There are two modes of operation, depending on the availability of negative triples:

1. Both positive and negative triples are provided via `X_pos` and `X_neg` respectively. The optimization is done using a second-order method (limited-memory BFGS), therefore no hyperparameter needs to be specified.
2. Only positive triples are provided, and the negative triples are generated by corruptions just like it is done in training or evaluation. The optimization is done using a first-order method (ADAM), therefore `batches_count` and `epochs` must be specified.

Calibration is highly dependent on the base rate of positive triples. Therefore, for mode (2) of operation, the user is required to provide the `positive_base_rate` argument. For mode (1), that can be inferred automatically by the relative sizes of the positive and negative sets, but the user can override that by providing a value to `positive_base_rate`.

Defining the positive base rate is the biggest challenge when calibrating without negatives. That depends on the user choice of which triples will be evaluated during test time. Let's take WN11 as an example: it has around 50% positives triples on both the validation set and test set, so naturally the positive base rate is 50%. However, should the user resample it to have 75% positives and 25% negatives, its previous calibration will be degraded. The user must recalibrate the model now with a 75% positive base rate. Therefore, this parameter depends on how the user handles the dataset and cannot be determined automatically or a priori.

Note: Incompatible with large graph mode (i.e. if `self.dealing_with_large_graphs=True`).

Note: [TC20] calibration experiments available here.

Parameters

- **X_pos** (*ndarray (shape [n, 3])*) – Numpy array of positive triples.
- **X_neg** (*ndarray (shape [n, 3])*) – Numpy array of negative triples.

  If *None*, the negative triples are generated via corruptions and the user must provide a positive base rate instead.
- **positive_base_rate** (*float*) – Base rate of positive statements.

  For example, if we assume there is a fifty-fifty chance of any query to be true, the base rate would be 50%.

  If `X_neg` is provided and this is *None*, the relative sizes of `X_pos` and `X_neg` will be used to determine the base rate. For example, if we have 50 positive triples and 200 negative triples, the positive base rate will be assumed to be 50/(50+200) = 1/5 = 0.2.

  This must be a value between 0 and 1.
- **batches_count** (*int*) – Number of batches to complete one epoch of the Platt scaling training. Only applies when `X_neg` is *None*.
- **epochs** (*int*) – Number of epochs used to train the Platt scaling model. Only applies when `X_neg` is *None*.

Examples

```
>>> import numpy as np
>>> from sklearn.metrics import brier_score_loss, log_loss
>>> from scipy.special import expit
>>>
>>> from ampligraph.datasets import load_wn11
>>> from ampligraph.latent_features.models import TransE
>>>
>>> X = load_wn11()
>>> X_valid_pos = X['valid'][X['valid_labels']]
>>> X_valid_neg = X['valid'][~X['valid_labels']]
>>>
>>> model = TransE(batches_count=64, seed=0, epochs=500, k=100, eta=20,
>>>                optimizer='adam', optimizer_params={'lr':0.0001},
>>>                loss='pairwise', verbose=True)
>>>
>>> model.fit(X['train'])
>>>
>>> # Raw scores
>>> scores = model.predict(X['test'])
>>>
>>> # Calibrate with positives and negatives
>>> model.calibrate(X_valid_pos, X_valid_neg, positive_base_rate=None)
>>> probas_pos_neg = model.predict_proba(X['test'])
>>>
>>> # Calibrate with just positives and base rate of 50%
>>> model.calibrate(X_valid_pos, positive_base_rate=0.5)
>>> probas_pos = model.predict_proba(X['test'])
>>>
>>> # Calibration evaluation with the Brier score loss (the smaller, the␣
↪better)
>>> print("Brier scores")
>>> print("Raw scores:", brier_score_loss(X['test_labels'], expit(scores)))
>>> print("Positive and negative calibration:", brier_score_loss(X['test_
↪labels'], probas_pos_neg))
>>> print("Positive only calibration:", brier_score_loss(X['test_labels'],␣
↪probas_pos))
Brier scores
Raw scores: 0.4925058891371126
Positive and negative calibration: 0.20434617882733366
Positive only calibration: 0.22597599585144656
```

predict_proba(*X*)

Predicts probabilities using the Platt scaling model (after calibration).

Model must be calibrated beforehand with the `calibrate` method.

Parameters **X** (*ndarray (shape [n, 3])*) – Numpy array of triples to be evaluated.

Returns **probas** – Probability of each triple to be true according to the Platt scaling calibration.

Return type ndarray (shape [n])

HolE

class ampligraph.latent_features.**HolE**(*k=100, eta=2, epochs=100, batches_count=100, seed=0, embedding_model_params={'corrupt_sides': ['s,o'], 'negative_corruption_entities': 'all'}, optimizer='adam', optimizer_params={'lr': 0.0005}, loss='nll', loss_params={}, regularizer=None, regularizer_params={}, initializer='xavier', initializer_params={'uniform': False}, verbose=False*)

Holographic Embeddings

The HolE model [NRP+16] as re-defined by Hayashi et al. [HS17]:

$$f_{HolE} = \frac{2}{n} f_{ComplEx}$$

Examples

```
>>> import numpy as np
>>> from ampligraph.latent_features import HolE
>>> model = HolE(batches_count=1, seed=555, epochs=100, k=10, eta=5,
>>>              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']]))
[[0.009254738], [0.00023370088]]
```

Methods

__init__([k, eta, epochs, batches_count, ...])	Initialize an EmbeddingModel
fit(X[, early_stopping, early_stopping_params])	Train a HolE model.
get_embeddings(entities[, embedding_type])	Get the embeddings of entities or relations.
get_hyperparameter_dict()	Returns hyperparameters of the model.
predict(X[, from_idx])	Predict the scores of triples using a trained embedding model.
calibrate(X_pos[, X_neg, ...])	Calibrate predictions
predict_proba(X)	Predicts probabilities using the Platt scaling model (after calibration).

__init__(*k=100, eta=2, epochs=100, batches_count=100, seed=0, embedding_model_params={'corrupt_sides': ['s,o'], 'negative_corruption_entities': 'all'}, optimizer='adam', optimizer_params={'lr': 0.0005}, loss='nll', loss_params={}, regularizer=None, regularizer_params={}, initializer='xavier', initializer_params={'uniform': False}, verbose=False*)

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:
  - **negative_corruption_entities** - Entities to be used for generation of corruptions while training. It can take the following values : `all` (default: all entities), `batch` (entities present in each batch), list of entities or an int (which indicates how many entities that should be used for corruption generation).
  - **corrupt_sides** : Specifies how to generate corruptions for training. Takes values *s*, *o*, *s+o* or any combination passed as a list.
- **optimizer** (*string*) – The optimizer used to minimize the loss function. Choose between 'sgd', 'adagrad', 'adam', 'momentum'.
- **optimizer_params** (*dict*) – Arguments specific to the optimizer, passed as a dictionary.

  Supported keys:
  - **'lr'** (float): learning rate (used by all the optimizers). Default: 0.1.
  - **'momentum'** (float): learning momentum (only used when `optimizer=momentum`). Default: 0.9.

  Example: `optimizer_params={'lr': 0.01}`
- **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.
  - `multiclass_nll` the model will use multiclass nll loss. Switch to multiclass loss defined in [aC15] by passing 'corrupt_sides' as ['s','o'] to embedding_model_params. To use loss defined in [KBK17] pass 'corrupt_sides' as 'o' to embedding_model_params.
- **loss_params** (*dict*) – Dictionary of loss-specific hyperparameters. See *loss functions* documentation for additional details.

  Example: `optimizer_params={'lr': 0.01}` if `loss='pairwise'`.
- **regularizer** (*string*) – The regularization strategy to use with the loss function.
  - `None`: the model will not use any regularizer (default)

- 'LP': the model will use L1, L2 or L3 based on the value of `regularizer_params['p']` (see below).

- **regularizer_params** (*dict*) – Dictionary of regularizer-specific hyperparameters. See the *regularizers* documentation for additional details.

  Example: `regularizer_params={'lambda': 1e-5, 'p': 2}` if `regularizer='LP'`.

- **initializer** (*string*) – The type of initializer to use.
  - `normal`: The embeddings will be initialized from a normal distribution
  - `uniform`: The embeddings will be initialized from a uniform distribution
  - `xavier`: The embeddings will be initialized using xavier strategy (default)

- **initializer_params** (*dict*) – Dictionary of initializer-specific hyperparameters. See the *initializer* documentation for additional details.

  Example: `initializer_params={'mean': 0, 'std': 0.001}` if `initializer='normal'`.

- **verbose** (*bool*) – Verbose mode.

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).

  If set to `True`, the training loop adopts the following early stopping heuristic:
  - The model will be trained regardless of early stopping for `burn_in` epochs.
  - Every `check_interval` epochs the method will compute the metric specified in `criteria`.

  If such metric decreases for `stop_interval` checks, we stop training early.

  Note the metric is computed on `x_valid`. This is usually a validation set that you held out.

  Also, because `criteria` is a ranking metric, it requires generating negatives. Entities used to generate corruptions can be specified, as long as the side(s) of a triple to corrupt. The method supports filtered metrics, by passing an array of positives to `x_filter`. This will be used to filter the negatives generated on the fly (i.e. the corruptions).

  Note: Keep in mind the early stopping criteria may introduce a certain overhead (caused by the metric computation). The goal is to strike a good trade-off between such overhead and saving training epochs.

  A common approach is to use MRR unfiltered:

```
early_stopping_params={x_valid=X['valid'], 'criteria': 'mrr'}
```

Note the size of validation set also contributes to such overhead. In most cases a smaller validation set would be enough.

- **early_stopping_params** (*dictionary*) – Dictionary of hyperparameters for the early stopping heuristics.

  The following string 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] : Positive triples to use as filter if a 'filtered' early stopping criteria is desired (i.e. filtered-MRR if 'criteria':'mrr'). Note this will affect training time (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)
  - **'corruption_entities'**: List of entities to be used for corruptions. If 'all', it uses all entities (default: 'all')
  - **'corrupt_side'**: Specifies which side to corrupt. 's', 'o', 's+o' (default)

  Example: `early_stopping_params={x_valid=X['valid'], 'criteria': 'mrr'}`

get_embeddings(*entities*, *embedding_type='entity'*)

Get the embeddings of entities or relations.

Note: Use *ampligraph.utils.create_tensorboard_visualizations()* to visualize the embeddings with TensorBoard.

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.
- **embedding_type** (*string*) – If 'entity', `entities` argument will be considered as a list of knowledge graph entities (i.e. nodes). If set to 'relation', they will be treated as relation types instead (i.e. predicates).

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

Return type ndarray, shape [n, k]

get_hyperparameter_dict()

Returns hyperparameters of the model.

Returns **hyperparam_dict** – Dictionary of hyperparameters that were used for training.

Return type dict

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

Predict the scores of triples using a trained embedding model. The function returns raw scores generated by the model.

Note: To obtain probability estimates, calibrate the model with *calibrate()*, then call *predict_proba()*.

Parameters

- **X** (*ndarray, shape [n, 3]*) – The triples to score.
- **from_idx** (*bool*) – If True, will skip conversion to internal IDs. (default: False).

Returns **scores_predict** – The predicted scores for input triples X.

Return type ndarray, shape [n]

calibrate(*X_pos*, *X_neg=None*, *positive_base_rate=None*, *batches_count=100*, *epochs=50*)

Calibrate predictions

The method implements the heuristics described in [TC20], using Platt scaling [P+99].

The calibrated predictions can be obtained with *predict_proba()* after calibration is done.

Ideally, calibration should be performed on a validation set that was not used to train the embeddings.

There are two modes of operation, depending on the availability of negative triples:

1. Both positive and negative triples are provided via `X_pos` and `X_neg` respectively. The optimization is done using a second-order method (limited-memory BFGS), therefore no hyperparameter needs to be specified.
2. Only positive triples are provided, and the negative triples are generated by corruptions just like it is done in training or evaluation. The optimization is done using a first-order method (ADAM), therefore `batches_count` and `epochs` must be specified.

Calibration is highly dependent on the base rate of positive triples. Therefore, for mode (2) of operation, the user is required to provide the `positive_base_rate` argument. For mode (1), that can be inferred automatically by the relative sizes of the positive and negative sets, but the user can override that by providing a value to `positive_base_rate`.

Defining the positive base rate is the biggest challenge when calibrating without negatives. That depends on the user choice of which triples will be evaluated during test time. Let's take WN11 as an example: it has around 50% positives triples on both the validation set and test set, so naturally the positive base rate is 50%. However, should the user resample it to have 75% positives and 25% negatives, its previous calibration will be degraded. The user must recalibrate the model now with a 75% positive base rate. Therefore, this parameter depends on how the user handles the dataset and cannot be determined automatically or a priori.

Note: Incompatible with large graph mode (i.e. if `self.dealing_with_large_graphs=True`).

Note: [TC20] calibration experiments available here.

Parameters

- **X_pos** (*ndarray (shape [n, 3])*) – Numpy array of positive triples.

- **X_neg** (*ndarray (shape [n, 3])*) – Numpy array of negative triples.

  If *None*, the negative triples are generated via corruptions and the user must provide a positive base rate instead.

- **positive_base_rate** (*float*) – Base rate of positive statements.

  For example, if we assume there is a fifty-fifty chance of any query to be true, the base rate would be 50%.

  If `X_neg` is provided and this is *None*, the relative sizes of `X_pos` and `X_neg` will be used to determine the base rate. For example, if we have 50 positive triples and 200 negative triples, the positive base rate will be assumed to be 50/(50+200) = 1/5 = 0.2.

  This must be a value between 0 and 1.

- **batches_count** (*int*) – Number of batches to complete one epoch of the Platt scaling training. Only applies when `X_neg` is *None*.

- **epochs** (*int*) – Number of epochs used to train the Platt scaling model. Only applies when `X_neg` is *None*.

Examples

```
>>> import numpy as np
>>> from sklearn.metrics import brier_score_loss, log_loss
>>> from scipy.special import expit
>>>
>>> from ampligraph.datasets import load_wn11
>>> from ampligraph.latent_features.models import TransE
>>>
>>> X = load_wn11()
>>> X_valid_pos = X['valid'][X['valid_labels']]
>>> X_valid_neg = X['valid'][~X['valid_labels']]
>>>
>>> model = TransE(batches_count=64, seed=0, epochs=500, k=100, eta=20,
>>>                optimizer='adam', optimizer_params={'lr':0.0001},
>>>                loss='pairwise', verbose=True)
>>>
>>> model.fit(X['train'])
>>>
>>> # Raw scores
>>> scores = model.predict(X['test'])
>>>
>>> # Calibrate with positives and negatives
>>> model.calibrate(X_valid_pos, X_valid_neg, positive_base_rate=None)
>>> probas_pos_neg = model.predict_proba(X['test'])
>>>
>>> # Calibrate with just positives and base rate of 50%
>>> model.calibrate(X_valid_pos, positive_base_rate=0.5)
>>> probas_pos = model.predict_proba(X['test'])
>>>
>>> # Calibration evaluation with the Brier score loss (the smaller, the␣
↪better)
>>> print("Brier scores")
>>> print("Raw scores:", brier_score_loss(X['test_labels'], expit(scores)))
>>> print("Positive and negative calibration:", brier_score_loss(X['test_
↪labels'], probas_pos_neg))
```

(continues on next page)

(continued from previous page)

```
>>> print("Positive only calibration:", brier_score_loss(X['test_labels'],␣
↪probas_pos))
Brier scores
Raw scores: 0.4925058891371126
Positive and negative calibration: 0.20434617882733366
Positive only calibration: 0.22597599585144656
```

predict_proba(*X*)

Predicts probabilities using the Platt scaling model (after calibration).

Model must be calibrated beforehand with the `calibrate` method.

Parameters **X** (*ndarray (shape [n, 3])*) – Numpy array of triples to be evaluated.

Returns **probas** – Probability of each triple to be true according to the Platt scaling calibration.

Return type ndarray (shape [n])

ConvE

class ampligraph.latent_features.**ConvE**(*k=100*, *eta=2*, *epochs=100*, *batches_count=100*, *seed=0*, *embedding_model_params={'conv_filters': 32*, *'conv_kernel_size': 3*, *'dropout_conv': 0.3*, *'dropout_dense': 0.2*, *'dropout_embed': 0.2*, *'use_batchnorm': True*, *'use_bias': True}*, *optimizer='adam'*, *optimizer_params={'lr': 0.0005}*, *loss='bce'*, *loss_params={'label_smoothing': 0.1*, *'label_weighting': False}*, *regularizer=None*, *regularizer_params={}*, *initializer='xavier'*, *initializer_params={'uniform': False}*, *low_memory=False*, *verbose=False*)

Convolutional 2D KG Embeddings

The ConvE model [DMSR18].

ConvE uses convolutional layers. g is a non-linear activation function, $*$ is the linear convolution operator, vec indicates 2D reshaping.

$$f_{ConvE} = \langle \sigma \left(vec \left(g \left([\overline{\mathbf{e}_s}; \overline{\mathbf{r}_p}] * \Omega \right) \right) \mathbf{W} \right) \mathbf{e}_o \rangle$$

Note: ConvE does not handle 's+o' corruptions currently, nor `large_graph` mode.

Examples

```
>>> import numpy as np
>>> from ampligraph.latent_features import ConvE
>>> model = ConvE(batches_count=1, seed=22, epochs=5, k=100)
>>>
>>> X = np.array([['a', 'y', 'b'],
>>>               ['b', 'y', 'a'],
>>>               ['a', 'y', 'c'],
```

(continues on next page)

(continued from previous page)

```
>>>                 ['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']]))
[0.42921206 0.38998795]
```

Attributes

__init__([k, eta, epochs, batches_count, ...])	Initialize a ConvE model
fit(X[, early_stopping, early_stopping_params])	Train a ConvE (with optional early stopping).
get_embeddings(entities[, embedding_type])	Get the embeddings of entities or relations.
get_hyperparameter_dict()	Returns hyperparameters of the model.
predict(X[, from_idx])	Predict the scores of triples using a trained embedding model.
calibrate(X_pos[, X_neg, ...])	Calibrate predictions
predict_proba(X)	Predicts probabilities using the Platt scaling model (after calibration).

__init__(*k=100*, *eta=2*, *epochs=100*, *batches_count=100*, *seed=0*, *embedding_model_params={'conv_filters': 32, 'conv_kernel_size': 3, 'dropout_conv': 0.3, 'dropout_dense': 0.2, 'dropout_embed': 0.2, 'use_batchnorm': True, 'use_bias': True}*, *optimizer='adam'*, *optimizer_params={'lr': 0.0005}*, *loss='bce'*, *loss_params={'label_smoothing': 0.1, 'label_weighting': False}*, *regularizer=None*, *regularizer_params={}*, *initializer='xavier'*, *initializer_params={'uniform': False}*, *low_memory=False*, *verbose=False*)

Initialize a ConvE model

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. Note: This parameter is not used in ConvE.
- **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*) – ConvE-specific hyperparams:
  - **conv_filters** (int): Number of convolution feature maps. Default: 32
  - **conv_kernel_size** (int): Convolution kernel size. Default: 3
  - **dropout_embed** (float|None): Dropout on the embedding layer. Default: 0.2
  - **dropout_conv** (float|None): Dropout on the convolution maps. Default: 0.3
  - **dropout_dense** (float|None): Dropout on the dense layer. Default: 0.2

  - **use_bias** (bool): Use bias layer. Default: True
  - **use_batchnorm** (bool): Use batch normalization after input, convolution, dense layers. Default: True

- **optimizer** (*string*) – The optimizer used to minimize the loss function. Choose between 'sgd', 'adagrad', 'adam', 'momentum'.
- **optimizer_params** (*dict*) – Arguments specific to the optimizer, passed as a dictionary.

  Supported keys:

  - **'lr'** (float): learning rate (used by all the optimizers). Default: 0.1.
  - **'momentum'** (float): learning momentum (only used when `optimizer=momentum`). Default: 0.9.

  Example: `optimizer_params={'lr': 0.01}`
- **loss** (*string*) – The type of loss function to use during training.
  - `bce` the model will use binary cross entropy loss function.
- **loss_params** (*dict*) – Dictionary of loss-specific hyperparameters. See *loss functions* documentation for additional details.

  Supported keys:

  - **'lr'** (float): learning rate (used by all the optimizers). Default: 0.1.
  - **'momentum'** (float): learning momentum (only used when `optimizer=momentum`). Default: 0.9.
  - **'label_smoothing'** (float): applies label smoothing to one-hot outputs. Default: 0.1.
  - **'label_weighting'** (bool): applies label weighting to one-hot outputs. Default: True

  Example: `optimizer_params={'lr': 0.01, 'label_smoothing': 0.1}`
- **regularizer** (*string*) – The regularization strategy to use with the loss function.
  - `None`: the model will not use any regularizer (default)
  - `LP`: the model will use L1, L2 or L3 based on the value of `regularizer_params['p']` (see below).
- **regularizer_params** (*dict*) – Dictionary of regularizer-specific hyperparameters. See the *regularizers* documentation for additional details.

  Example: `regularizer_params={'lambda': 1e-5, 'p': 2}` if `regularizer='LP'`.
- **initializer** (*string*) – The type of initializer to use.
  - `normal`: The embeddings will be initialized from a normal distribution
  - `uniform`: The embeddings will be initialized from a uniform distribution
  - `xavier`: The embeddings will be initialized using xavier strategy (default)

- **initializer_params** (*dict*) – Dictionary of initializer-specific hyperparameters. See the *initializer* documentation for additional details.

  Example: `initializer_params={'mean': 0, 'std': 0.001}` if `initializer='normal'`.

- **verbose** (*bool*) – Verbose mode.
- **low_memory** (*bool*) – Train ConvE with a (slower) low_memory option. If MemoryError is still encountered, try raising the batches_count value. Default: False.

fit (*X*, *early_stopping=False*, *early_stopping_params={}*)

Train a ConvE (with optional early stopping).

The model is trained on a training set X using the training protocol described in [DMSR18].

Parameters

- **X** (*ndarray (shape [n, 3]) or object of ConvEDatasetAdapter*) – Numpy array of training triples OR handle of Dataset adapter which would help retrieve data.
- **early_stopping** (*bool*) – Flag to enable early stopping (default:`False`)
- **early_stopping_params** (*dictionary*) – Dictionary of hyperparameters for the early stopping heuristics.

  The following string keys are supported:

  - **'x_valid': ndarray (shape [n, 3]) or object of AmpligraphDatasetAdapter :** Numpy array of validation triples OR handle of Dataset adapter which would help retrieve data.
  - **'criteria'**: string : criteria for early stopping 'hits10', 'hits3', 'hits1' or 'mrr'(default).
  - **'x_filter': ndarray, shape [n, 3]** [Positive triples to use as filter if a 'filtered' early] stopping criteria is desired (i.e. filtered-MRR if 'criteria':'mrr'). Note this will affect training time (no filter by default). If the filter has already been set in the adapter, pass True
  - **'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)
  - **'corruption_entities'**: List of entities to be used for corruptions. If 'all', it uses all entities (default: 'all')
  - **'corrupt_side': Specifies which side to corrupt. 's', 'o', 's,o' (default). Note: ConvE does not** currently support 's+o' evaluation mode.

  Example: `early_stopping_params={x_valid=X['valid'], 'criteria': 'mrr'}`

get_embeddings (*entities*, *embedding_type='entity'*)

Get the embeddings of entities or relations.

Note: Use *ampligraph.utils.create_tensorboard_visualizations()* to visualize the embeddings with TensorBoard.

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.
- **embedding_type** (*string*) – If 'entity', `entities` argument will be considered as a list of knowledge graph entities (i.e. nodes). If set to 'relation', they will be treated as relation types instead (i.e. predicates).

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

Return type ndarray, shape [n, k]

get_hyperparameter_dict()
Returns hyperparameters of the model.

Returns **hyperparam_dict** – Dictionary of hyperparameters that were used for training.

Return type dict

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

Predict the scores of triples using a trained embedding model. The function returns raw scores generated by the model.

Note: To obtain probability estimates, calibrate the model with *calibrate()*, then call *predict_proba()*.

Parameters

- **X** (*ndarray, shape [n, 3]*) – The triples to score.
- **from_idx** (*bool*) – If True, will skip conversion to internal IDs. (default: False).

Returns **scores_predict** – The predicted scores for input triples X.

Return type ndarray, shape [n]

calibrate(*X_pos*, *X_neg=None*, *positive_base_rate=None*, *batches_count=100*, *epochs=50*)
Calibrate predictions

The method implements the heuristics described in [TC20], using Platt scaling [P+99].

The calibrated predictions can be obtained with *predict_proba()* after calibration is done.

Ideally, calibration should be performed on a validation set that was not used to train the embeddings.

There are two modes of operation, depending on the availability of negative triples:

1. Both positive and negative triples are provided via `X_pos` and `X_neg` respectively. The optimization is done using a second-order method (limited-memory BFGS), therefore no hyperparameter needs to be specified.

2. Only positive triples are provided, and the negative triples are generated by corruptions just like it is done in training or evaluation. The optimization is done using a first-order method (ADAM), therefore `batches_count` and `epochs` must be specified.

Calibration is highly dependent on the base rate of positive triples. Therefore, for mode (2) of operation, the user is required to provide the `positive_base_rate` argument. For mode (1), that can be inferred automatically by the relative sizes of the positive and negative sets, but the user can override that by providing a value to `positive_base_rate`.

Defining the positive base rate is the biggest challenge when calibrating without negatives. That depends on the user choice of which triples will be evaluated during test time. Let's take WN11 as an example: it has around 50% positives triples on both the validation set and test set, so naturally the positive base rate is 50%. However, should the user resample it to have 75% positives and 25% negatives, its previous calibration will be degraded. The user must recalibrate the model now with a 75% positive base rate. Therefore, this parameter depends on how the user handles the dataset and cannot be determined automatically or a priori.

Note: Incompatible with large graph mode (i.e. if `self.dealing_with_large_graphs=True`).

Note: [TC20] calibration experiments available here.

Parameters

- **X_pos** (*ndarray (shape [n, 3])*) – Numpy array of positive triples.
- **X_neg** (*ndarray (shape [n, 3])*) – Numpy array of negative triples.

  If *None*, the negative triples are generated via corruptions and the user must provide a positive base rate instead.

- **positive_base_rate** (*float*) – Base rate of positive statements.

  For example, if we assume there is a fifty-fifty chance of any query to be true, the base rate would be 50%.

  If `X_neg` is provided and this is *None*, the relative sizes of `X_pos` and `X_neg` will be used to determine the base rate. For example, if we have 50 positive triples and 200 negative triples, the positive base rate will be assumed to be 50/(50+200) = 1/5 = 0.2.

  This must be a value between 0 and 1.

- **batches_count** (*int*) – Number of batches to complete one epoch of the Platt scaling training. Only applies when `X_neg` is *None*.
- **epochs** (*int*) – Number of epochs used to train the Platt scaling model. Only applies when `X_neg` is *None*.

Examples

```
>>> import numpy as np
>>> from sklearn.metrics import brier_score_loss, log_loss
>>> from scipy.special import expit
>>>
>>> from ampligraph.datasets import load_wn11
>>> from ampligraph.latent_features.models import TransE
>>>
>>> X = load_wn11()
>>> X_valid_pos = X['valid'][X['valid_labels']]
>>> X_valid_neg = X['valid'][~X['valid_labels']]
>>>
>>> model = TransE(batches_count=64, seed=0, epochs=500, k=100, eta=20,
>>>                optimizer='adam', optimizer_params={'lr':0.0001},
>>>                loss='pairwise', verbose=True)
>>>
>>> model.fit(X['train'])
>>>
>>> # Raw scores
>>> scores = model.predict(X['test'])
>>>
>>> # Calibrate with positives and negatives
>>> model.calibrate(X_valid_pos, X_valid_neg, positive_base_rate=None)
>>> probas_pos_neg = model.predict_proba(X['test'])
>>>
>>> # Calibrate with just positives and base rate of 50%
>>> model.calibrate(X_valid_pos, positive_base_rate=0.5)
>>> probas_pos = model.predict_proba(X['test'])
>>>
>>> # Calibration evaluation with the Brier score loss (the smaller, the␣
↪better)
>>> print("Brier scores")
>>> print("Raw scores:", brier_score_loss(X['test_labels'], expit(scores)))
>>> print("Positive and negative calibration:", brier_score_loss(X['test_
↪labels'], probas_pos_neg))
>>> print("Positive only calibration:", brier_score_loss(X['test_labels'],␣
↪probas_pos))
Brier scores
Raw scores: 0.4925058891371126
Positive and negative calibration: 0.20434617882733366
Positive only calibration: 0.22597599585144656
```

predict_proba(*X*)

Predicts probabilities using the Platt scaling model (after calibration).

Model must be calibrated beforehand with the `calibrate` method.

Parameters **X** (*ndarray (shape [n, 3])*) – Numpy array of triples to be evaluated.

Returns **probas** – Probability of each triple to be true according to the Platt scaling calibration.

Return type ndarray (shape [n])

ConvKB

class ampligraph.latent_features.**ConvKB**(*k=100, eta=2, epochs=100, batches_count=100, seed=0, embedding_model_params={'dropout': 0.1, 'filter_sizes': [1], 'num_filters': 32}, optimizer='adam', optimizer_params={'lr': 0.0005}, loss='nll', loss_params={}, regularizer=None, regularizer_params={}, initializer='xavier', initializer_params={'uniform': False}, large_graphs=False, verbose=False*)

Convolution-based model

The ConvKB model [NNNP18]:

$$f_{ConvKB} = concat\left(g\left(\left[\mathbf{e}_s, \mathbf{r}_p, \mathbf{e}_o\right]\right) * \Omega\right) \cdot W$$

where g is a non-linear function, $*$ is the convolution operator, $\cdot$ is the dot product, $concat$ is the concatenation operator and Ω is a set of filters.

Note: The evaluation protocol implemented in *ampligraph.evaluation.evaluate_performance()* assigns the worst rank to a positive test triple in case of a tie with negatives. This is the agreed upon behaviour in literature. The original ConvKB implementation [NNNP18] assigns instead the top rank, hence leading to results which are not directly comparable with literature . We report results obtained with the agreed-upon protocol (tie=worst rank). Note that under these conditions the model *does not reach the state-of-the-art results claimed in the original paper*.

Examples

```
>>> from ampligraph.latent_features import ConvKB
>>> from ampligraph.datasets import load_wn18
>>> model = ConvKB(batches_count=2, seed=22, epochs=1, k=10, eta=1,
>>>                embedding_model_params={'num_filters': 32, 'filter_sizes': [1],
>>>                                        'dropout': 0.1},
>>>                optimizer='adam', optimizer_params={'lr': 0.001},
>>>                loss='pairwise', loss_params={}, verbose=True)
>>>
>>> X = load_wn18()
>>>
>>> model.fit(X['train'])
>>>
>>> print(model.predict(X['test'][:5]))
[[0.2803744], [0.0866661], [0.012815937], [-0.004235901], [-0.010947697]]
```

Methods

__init__([k, eta, epochs, batches_count, ...])	Initialize an EmbeddingModel
fit(X[, early_stopping, early_stopping_params])	Train a ConvKB model (with optional early stopping).
get_embeddings(entities[, embedding_type])	Get the embeddings of entities or relations.
get_hyperparameter_dict()	Returns hyperparameters of the model.

Continued on next page

Table 10 – continued from previous page

predict(X[, from_idx])	Predict the scores of triples using a trained embedding model.
calibrate(X_pos[, X_neg, ...])	Calibrate predictions
predict_proba(X)	Predicts probabilities using the Platt scaling model (after calibration).

__init__ (*k=100, eta=2, epochs=100, batches_count=100, seed=0, embedding_model_params={'dropout': 0.1, 'filter_sizes': [1], 'num_filters': 32}, optimizer='adam', optimizer_params={'lr': 0.0005}, loss='nll', loss_params={}, regularizer=None, regularizer_params={}, initializer='xavier', initializer_params={'uniform': False}, large_graphs=False, verbose=False*)

Initialize an EmbeddingModel

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*) – ConvKB-specific hyperparams: - **num_filters** - Number of feature maps per convolution kernel. Default: 32 - **filter_sizes** - List of convolution kernel sizes. Default: [1] - **dropout** - Dropout on the embedding layer. Default: 0.0
- **optimizer** (*string*) – The optimizer used to minimize the loss function. Choose between 'sgd', 'adagrad', 'adam', 'momentum'.
- **optimizer_params** (*dict*) – Arguments specific to the optimizer, passed as a dictionary.

  Supported keys:

  - **'lr'** (float): learning rate (used by all the optimizers). Default: 0.1.
  - **'momentum'** (float): learning momentum (only used when `optimizer=momentum`). Default: 0.9.

  Example: `optimizer_params={'lr': 0.01}`
- **loss** (*string*) – The type of loss function to use during training.
- **loss_params** (*dict*) – Dictionary of loss-specific hyperparameters. See *loss functions* documentation for additional details.

  Supported keys:

  - **'lr'** (float): learning rate (used by all the optimizers). Default: 0.1.
  - **'momentum'** (float): learning momentum (only used when `optimizer=momentum`). Default: 0.9.

  Example: `optimizer_params={'lr': 0.01, 'label_smoothing': 0.1}`

- **regularizer** (*string*) – The regularization strategy to use with the loss function.
  - `None`: the model will not use any regularizer (default)
  - `LP`: the model will use L1, L2 or L3 based on the value of `regularizer_params['p']` (see below).
- **regularizer_params** (*dict*) – Dictionary of regularizer-specific hyperparameters. See the *regularizers* documentation for additional details.

  Example: `regularizer_params={'lambda': 1e-5, 'p': 2}` if `regularizer='LP'`.
- **initializer** (*string*) – The type of initializer to use.
  - `normal`: The embeddings will be initialized from a normal distribution
  - `uniform`: The embeddings will be initialized from a uniform distribution
  - `xavier`: The embeddings will be initialized using xavier strategy (default)
- **initializer_params** (*dict*) – Dictionary of initializer-specific hyperparameters. See the *initializer* documentation for additional details.

  Example: `initializer_params={'mean': 0, 'std': 0.001}` if `initializer='normal'`.
- **large_graphs** (*bool*) – Avoid loading entire dataset onto GPU when dealing with large graphs.
- **verbose** (*bool*) – Verbose mode.

fit (*X*, *early_stopping=False*, *early_stopping_params={}*)

Train a ConvKB model (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).

  If set to `True`, the training loop adopts the following early stopping heuristic:
  - The model will be trained regardless of early stopping for `burn_in` epochs.
  - Every `check_interval` epochs the method will compute the metric specified in `criteria`.

  If such metric decreases for `stop_interval` checks, we stop training early.

  Note the metric is computed on `x_valid`. This is usually a validation set that you held out.

  Also, because `criteria` is a ranking metric, it requires generating negatives. Entities used to generate corruptions can be specified, as long as the side(s) of a triple to corrupt. The method supports filtered metrics, by passing an array of positives to `x_filter`. This will be used to filter the negatives generated on the fly (i.e. the corruptions).

  Note: Keep in mind the early stopping criteria may introduce a certain overhead (caused by the metric computation). The goal is to strike a good trade-off between such overhead and saving training epochs.

A common approach is to use MRR unfiltered:

```
early_stopping_params={x_valid=X['valid'], 'criteria':
'mrr'}
```

Note the size of validation set also contributes to such overhead. In most cases a smaller validation set would be enough.

- **early_stopping_params** (*dictionary*) – Dictionary of hyperparameters for the early stopping heuristics.

  The following string 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] : Positive triples to use as filter if a 'filtered' early stopping criteria is desired (i.e. filtered-MRR if 'criteria':'mrr'). Note this will affect training time (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)
  - **'corruption_entities'**: List of entities to be used for corruptions. If 'all', it uses all entities (default: 'all')
  - **'corrupt_side'**: Specifies which side to corrupt. 's', 'o', 's+o' (default)

  Example: `early_stopping_params={x_valid=X['valid'], 'criteria': 'mrr'}`

get_embeddings (*entities*, *embedding_type='entity'*)

Get the embeddings of entities or relations.

Note: Use *ampligraph.utils.create_tensorboard_visualizations()* to visualize the embeddings with TensorBoard.

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.
- **embedding_type** (*string*) – If 'entity', `entities` argument will be considered as a list of knowledge graph entities (i.e. nodes). If set to 'relation', they will be treated as relation types instead (i.e. predicates).

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

Return type ndarray, shape [n, k]

get_hyperparameter_dict()
Returns hyperparameters of the model.

Returns **hyperparam_dict** – Dictionary of hyperparameters that were used for training.

Return type dict

predict(*X*, *from_idx=False*)
Predict the scores of triples using a trained embedding model. The function returns raw scores generated by the model.

Note: To obtain probability estimates, calibrate the model with *calibrate()*, then call *predict_proba()*.

Parameters

- **X** (*ndarray, shape [n, 3]*) – The triples to score.
- **from_idx** (*bool*) – If True, will skip conversion to internal IDs. (default: False).

Returns **scores_predict** – The predicted scores for input triples X.

Return type ndarray, shape [n]

calibrate(*X_pos*, *X_neg=None*, *positive_base_rate=None*, *batches_count=100*, *epochs=50*)
Calibrate predictions

The method implements the heuristics described in [TC20], using Platt scaling [P+99].

The calibrated predictions can be obtained with *predict_proba()* after calibration is done.

Ideally, calibration should be performed on a validation set that was not used to train the embeddings.

There are two modes of operation, depending on the availability of negative triples:

1. Both positive and negative triples are provided via `X_pos` and `X_neg` respectively. The optimization is done using a second-order method (limited-memory BFGS), therefore no hyperparameter needs to be specified.
2. Only positive triples are provided, and the negative triples are generated by corruptions just like it is done in training or evaluation. The optimization is done using a first-order method (ADAM), therefore `batches_count` and `epochs` must be specified.

Calibration is highly dependent on the base rate of positive triples. Therefore, for mode (2) of operation, the user is required to provide the `positive_base_rate` argument. For mode (1), that can be inferred automatically by the relative sizes of the positive and negative sets, but the user can override that by providing a value to `positive_base_rate`.

Defining the positive base rate is the biggest challenge when calibrating without negatives. That depends on the user choice of which triples will be evaluated during test time. Let's take WN11 as an example: it has around 50% positives triples on both the validation set and test set, so naturally the positive base rate is 50%. However, should the user resample it to have 75% positives and 25% negatives, its previous calibration will be degraded. The user must recalibrate the model now with a 75% positive base rate. Therefore, this parameter depends on how the user handles the dataset and cannot be determined automatically or a priori.

Note: Incompatible with large graph mode (i.e. if `self.dealing_with_large_graphs=True`).

Note: [TC20] calibration experiments available here.

Parameters

- **X_pos** (*ndarray (shape [n, 3])*) – Numpy array of positive triples.
- **X_neg** (*ndarray (shape [n, 3])*) – Numpy array of negative triples.

  If *None*, the negative triples are generated via corruptions and the user must provide a positive base rate instead.
- **positive_base_rate** (*float*) – Base rate of positive statements.

  For example, if we assume there is a fifty-fifty chance of any query to be true, the base rate would be 50%.

  If `X_neg` is provided and this is *None*, the relative sizes of `X_pos` and `X_neg` will be used to determine the base rate. For example, if we have 50 positive triples and 200 negative triples, the positive base rate will be assumed to be 50/(50+200) = 1/5 = 0.2.

  This must be a value between 0 and 1.
- **batches_count** (*int*) – Number of batches to complete one epoch of the Platt scaling training. Only applies when `X_neg` is *None*.
- **epochs** (*int*) – Number of epochs used to train the Platt scaling model. Only applies when `X_neg` is *None*.

Examples

```
>>> import numpy as np
>>> from sklearn.metrics import brier_score_loss, log_loss
>>> from scipy.special import expit
>>>
>>> from ampligraph.datasets import load_wn11
>>> from ampligraph.latent_features.models import TransE
>>>
>>> X = load_wn11()
>>> X_valid_pos = X['valid'][X['valid_labels']]
>>> X_valid_neg = X['valid'][~X['valid_labels']]
>>>
>>> model = TransE(batches_count=64, seed=0, epochs=500, k=100, eta=20,
>>>                optimizer='adam', optimizer_params={'lr':0.0001},
>>>                loss='pairwise', verbose=True)
>>>
>>> model.fit(X['train'])
>>>
>>> # Raw scores
>>> scores = model.predict(X['test'])
>>>
>>> # Calibrate with positives and negatives
>>> model.calibrate(X_valid_pos, X_valid_neg, positive_base_rate=None)
>>> probas_pos_neg = model.predict_proba(X['test'])
>>>
>>> # Calibrate with just positives and base rate of 50%
>>> model.calibrate(X_valid_pos, positive_base_rate=0.5)
```

(continues on next page)

(continued from previous page)

```
>>> probas_pos = model.predict_proba(X['test'])
>>>
>>> # Calibration evaluation with the Brier score loss (the smaller, the␣
↪better)
>>> print("Brier scores")
>>> print("Raw scores:", brier_score_loss(X['test_labels'], expit(scores)))
>>> print("Positive and negative calibration:", brier_score_loss(X['test_
↪labels'], probas_pos_neg))
>>> print("Positive only calibration:", brier_score_loss(X['test_labels'],␣
↪probas_pos))
Brier scores
Raw scores: 0.4925058891371126
Positive and negative calibration: 0.20434617882733366
Positive only calibration: 0.22597599585144656
```

predict_proba(*X*)

Predicts probabilities using the Platt scaling model (after calibration).

Model must be calibrated beforehand with the `calibrate` method.

Parameters **X** (*ndarray (shape [n, 3])*) – Numpy array of triples to be evaluated.

Returns **probas** – Probability of each triple to be true according to the Platt scaling calibration.

Return type ndarray (shape [n])

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.
Initializer([initializer_params, verbose, seed])	Abstract class for initializer .

EmbeddingModel

class ampligraph.latent_features.**EmbeddingModel**(*k=100*, *eta=2*, *epochs=100*, *batches_count=100*, *seed=0*, *embedding_model_params={}*, *optimizer='adam'*, *optimizer_params={'lr': 0.0005}*, *loss='nll'*, *loss_params={}*, *regularizer=None*, *regularizer_params={}*, *initializer='xavier'*, *initializer_params={'uniform': False}*, *large_graphs=False*, *verbose=False*)

Abstract class for embedding models

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

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[, embedding_type])	Get the embeddings of entities or relations.
get_hyperparameter_dict()	Returns hyperparameters of the model.
predict(X[, from_idx])	Predict the scores of triples using a trained embedding model.
calibrate(X_pos[, X_neg, ...])	Calibrate predictions
predict_proba(X)	Predicts probabilities using the Platt scaling model (after calibration).
_fn(e_s, e_p, e_o)	The scoring function of the model.
_initialize_parameters()	Initialize parameters of the model.
_get_model_loss(dataset_iterator)	Get the current loss including loss due to regularization.
get_embedding_model_params(output_dict)	Save the model parameters in the dictionary.
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)	Performs regular validation checks and stop early if the criteria is achieved.
configure_evaluation_protocol([config])	Set the configuration for evaluation
set_filter_for_eval()	Configures to use filter
_initialize_eval_graph([mode])	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='adam'*, *optimizer_params={'lr': 0.0005}*, *loss='nll'*, *loss_params={}*, *regularizer=None*, *regularizer_params={}*, *initializer='xavier'*, *initializer_params={'uniform': False}*, *large_graphs=False*, *verbose=False*)

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*) – Model-specific hyperparams, passed to the model as a dictionary. Refer to model-specific documentation for details.
- **optimizer** (*string*) – The optimizer used to minimize the loss function. Choose between 'sgd', 'adagrad', 'adam', 'momentum'.
- **optimizer_params** (*dict*) – Arguments specific to the optimizer, passed as a dictionary.

  Supported keys:

  - **'lr'** (float): learning rate (used by all the optimizers). Default: 0.1.
  - **'momentum'** (float): learning momentum (only used when `optimizer=momentum`). Default: 0.9.

  Example: `optimizer_params={'lr': 0.01}`
- **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.
  - `multiclass_nll` the model will use multiclass nll loss. Switch to multiclass loss defined in [aC15] by passing 'corrupt_side' as ['s','o'] to embedding_model_params. To use loss defined in [KBK17] pass 'corrupt_side' as 'o' to embedding_model_params.
- **loss_params** (*dict*) – Dictionary of loss-specific hyperparameters. See *loss functions* documentation for additional details.

  Example: `optimizer_params={'lr': 0.01}` if `loss='pairwise'`.
- **regularizer** (*string*) – The regularization strategy to use with the loss function.
  - `None`: the model will not use any regularizer (default)
  - `LP`: the model will use L1, L2 or L3 based on the value of `regularizer_params['p']` (see below).
- **regularizer_params** (*dict*) – Dictionary of regularizer-specific hyperparameters. See the *regularizers* documentation for additional details.

  Example: `regularizer_params={'lambda': 1e-5, 'p': 2}` if `regularizer='LP'`.

- **initializer** (*string*) – The type of initializer to use.
  - `normal`: The embeddings will be initialized from a normal distribution
  - `uniform`: The embeddings will be initialized from a uniform distribution
  - `xavier`: The embeddings will be initialized using xavier strategy (default)
- **initializer_params** (*dict*) – Dictionary of initializer-specific hyperparameters. See the *initializer* documentation for additional details.

  Example: `initializer_params={'mean': 0, 'std': 0.001}` if `initializer='normal'`.
- **large_graphs** (*bool*) – Avoid loading entire dataset onto GPU when dealing with large graphs.
- **verbose** (*bool*) – Verbose mode.

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]) or object of AmpligraphDatasetAdapter*) – Numpy array of training triples OR handle of Dataset adapter which would help retrieve data.
- **early_stopping** (*bool*) – Flag to enable early stopping (default:`False`)
- **early_stopping_params** (*dictionary*) – Dictionary of hyperparameters for the early stopping heuristics.

  The following string keys are supported:
  - **'x_valid': ndarray (shape [n, 3]) or object of AmpligraphDatasetAdapter :** Numpy array of validation triples OR handle of Dataset adapter which would help retrieve data.
  - **'criteria'**: string : criteria for early stopping 'hits10', 'hits3', 'hits1' or 'mrr'(default).
  - **'x_filter': ndarray, shape [n, 3]** [Positive triples to use as filter if a 'filtered' early] stopping criteria is desired (i.e. filtered-MRR if 'criteria':'mrr'). Note this will affect training time (no filter by default). If the filter has already been set in the adapter, pass True
  - **'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)
  - **'corruption_entities'**: List of entities to be used for corruptions. If 'all', it uses all entities (default: 'all')
  - **'corrupt_side'**: Specifies which side to corrupt. 's', 'o', 's+o', 's,o' (default)

  Example: `early_stopping_params={x_valid=X['valid'], 'criteria': 'mrr'}`

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

Note: Use *ampligraph.utils.create_tensorboard_visualizations()* to visualize the embeddings with TensorBoard.

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.
- **embedding_type** (*string*) – If 'entity', `entities` argument will be considered as a list of knowledge graph entities (i.e. nodes). If set to 'relation', they will be treated as relation types instead (i.e. predicates).

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

Return type ndarray, shape [n, k]

get_hyperparameter_dict()
Returns hyperparameters of the model.

Returns **hyperparam_dict** – Dictionary of hyperparameters that were used for training.

Return type dict

predict(*X*, *from_idx=False*)
Predict the scores of triples using a trained embedding model. The function returns raw scores generated by the model.

Note: To obtain probability estimates, calibrate the model with *calibrate()*, then call *predict_proba()*.

Parameters

- **X** (*ndarray, shape [n, 3]*) – The triples to score.
- **from_idx** (*bool*) – If True, will skip conversion to internal IDs. (default: False).

Returns **scores_predict** – The predicted scores for input triples X.

Return type ndarray, shape [n]

calibrate(*X_pos*, *X_neg=None*, *positive_base_rate=None*, *batches_count=100*, *epochs=50*)
Calibrate predictions

The method implements the heuristics described in [TC20], using Platt scaling [P+99].

The calibrated predictions can be obtained with *predict_proba()* after calibration is done.

Ideally, calibration should be performed on a validation set that was not used to train the embeddings.

There are two modes of operation, depending on the availability of negative triples:

1. Both positive and negative triples are provided via `X_pos` and `X_neg` respectively. The optimization is done using a second-order method (limited-memory BFGS), therefore no hyperparameter needs to be specified.

2. Only positive triples are provided, and the negative triples are generated by corruptions just like it is done in training or evaluation. The optimization is done using a first-order method (ADAM), therefore `batches_count` and `epochs` must be specified.

Calibration is highly dependent on the base rate of positive triples. Therefore, for mode (2) of operation, the user is required to provide the `positive_base_rate` argument. For mode (1), that can be inferred automatically by the relative sizes of the positive and negative sets, but the user can override that by providing a value to `positive_base_rate`.

Defining the positive base rate is the biggest challenge when calibrating without negatives. That depends on the user choice of which triples will be evaluated during test time. Let's take WN11 as an example: it has around 50% positives triples on both the validation set and test set, so naturally the positive base rate is 50%. However, should the user resample it to have 75% positives and 25% negatives, its previous calibration will be degraded. The user must recalibrate the model now with a 75% positive base rate. Therefore, this parameter depends on how the user handles the dataset and cannot be determined automatically or a priori.

Note: Incompatible with large graph mode (i.e. if `self.dealing_with_large_graphs=True`).

Note: [TC20] calibration experiments available here.

Parameters

- **X_pos** (*ndarray (shape [n, 3])*) – Numpy array of positive triples.
- **X_neg** (*ndarray (shape [n, 3])*) – Numpy array of negative triples.

  If *None*, the negative triples are generated via corruptions and the user must provide a positive base rate instead.
- **positive_base_rate** (*float*) – Base rate of positive statements.

  For example, if we assume there is a fifty-fifty chance of any query to be true, the base rate would be 50%.

  If `X_neg` is provided and this is *None*, the relative sizes of `X_pos` and `X_neg` will be used to determine the base rate. For example, if we have 50 positive triples and 200 negative triples, the positive base rate will be assumed to be 50/(50+200) = 1/5 = 0.2.

  This must be a value between 0 and 1.
- **batches_count** (*int*) – Number of batches to complete one epoch of the Platt scaling training. Only applies when `X_neg` is *None*.
- **epochs** (*int*) – Number of epochs used to train the Platt scaling model. Only applies when `X_neg` is *None*.

Examples

```
>>> import numpy as np
>>> from sklearn.metrics import brier_score_loss, log_loss
>>> from scipy.special import expit
>>>
>>> from ampligraph.datasets import load_wn11
>>> from ampligraph.latent_features.models import TransE
>>>
>>> X = load_wn11()
>>> X_valid_pos = X['valid'][X['valid_labels']]
>>> X_valid_neg = X['valid'][~X['valid_labels']]
>>>
>>> model = TransE(batches_count=64, seed=0, epochs=500, k=100, eta=20,
>>>                optimizer='adam', optimizer_params={'lr':0.0001},
>>>                loss='pairwise', verbose=True)
>>>
>>> model.fit(X['train'])
>>>
>>> # Raw scores
>>> scores = model.predict(X['test'])
>>>
>>> # Calibrate with positives and negatives
>>> model.calibrate(X_valid_pos, X_valid_neg, positive_base_rate=None)
>>> probas_pos_neg = model.predict_proba(X['test'])
>>>
>>> # Calibrate with just positives and base rate of 50%
>>> model.calibrate(X_valid_pos, positive_base_rate=0.5)
>>> probas_pos = model.predict_proba(X['test'])
>>>
>>> # Calibration evaluation with the Brier score loss (the smaller, the␣
↪better)
>>> print("Brier scores")
>>> print("Raw scores:", brier_score_loss(X['test_labels'], expit(scores)))
>>> print("Positive and negative calibration:", brier_score_loss(X['test_
↪labels'], probas_pos_neg))
>>> print("Positive only calibration:", brier_score_loss(X['test_labels'],␣
↪probas_pos))
Brier scores
Raw scores: 0.4925058891371126
Positive and negative calibration: 0.20434617882733366
Positive only calibration: 0.22597599585144656
```

predict_proba(*X*)

Predicts probabilities using the Platt scaling model (after calibration).

Model must be calibrated beforehand with the `calibrate` method.

Parameters **X** (*ndarray (shape [n, 3])*) – Numpy array of triples to be evaluated.

Returns **probas** – Probability of each triple to be true according to the Platt scaling calibration.

Return type ndarray (shape [n])

abstract _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). If the graph is large, then it loads only the required entity embeddings (max:batch_size*2) and all relation embeddings. Overload this function if the parameters needs to be initialized differently.

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

Parameters **dataset_iterator** (*tf.data.Iterator*) – Dataset iterator.

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). This function also set's the evaluation mode to do lazy loading of variables based on the number of distinct entities present in the graph.

_initialize_early_stopping()
Initializes and creates evaluation graph for early stopping.

_perform_early_stopping_test(*epoch*)
Performs regular validation checks and stop early if the criteria is achieved.

Parameters **epoch** (*int*) – current training epoch.

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

Return type bool

configure_evaluation_protocol(*config=None*)
Set the configuration for evaluation

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

- **corruption_entities**: List of entities to be used for corruptions. If `all`, it uses all entities (default: `all`)
- **corrupt_side**: Specifies which side to corrupt. `s`, `o`, `s+o`, `s,o` (default) In 's,o' mode subject and object corruptions are generated at once but ranked separately for speed up (default: False).

set_filter_for_eval()
Configures to use filter

_initialize_eval_graph(*mode='test'*)
Initialize the evaluation graph.

Parameters **mode** (*string*) – Indicates which data generator to use.

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

Initializer

class ampligraph.latent_features.**Initializer**(*initializer_params={}*, *verbose=True*, *seed=0*)

Abstract class for initializer .

Methods

__init__([initializer_params, verbose, seed])	Initialize the Class
_init_hyperparams(hyperparam_dict)	Initializes the hyperparameters.
get_tf_initializer()	Create a tensorflow node for initializer
get_np_initializer(in_shape, out_shape)	Create an initialized numpy array
_display_params()	Display the parameter values

__init__(*initializer_params={}*, *verbose=True*, *seed=0*)

Initialize the Class

Parameters

- **initializer_params** (*dict*) – dictionary of hyperparams that would be used by the initializer.
- **verbose** (*bool*) – set/reset verbose mode
- **seed** (*int/np.random.RandomState*) – random state for random number generator

_init_hyperparams(*hyperparam_dict*)

Initializes the hyperparameters.

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

get_tf_initializer()

Create a tensorflow node for initializer

Returns **initializer_instance**

Return type An Initializer instance.

get_np_initializer(*in_shape*, *out_shape*)

Create an initialized numpy array

Parameters

- **in_shape** (*int*) – number of inputs to the layer (fan in)
- **out_shape** (*int*) – number of outputs of the layer (fan out)

Returns **initialized_values** – Initialized weights

Return type n-d array

_display_params()

Display the parameter values

Scoring functions

Existing models propose scoring functions that combine the embeddings $\mathbf{e}_s, \mathbf{r}_p, \mathbf{e}_o \in \mathcal{R}^k$ of the subject, predicate, and object of a triple $t = (s, p, o)$ 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 $||\cdot||$:

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

- *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 (denoted by $\otimes$):

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

- *ConvE* [DMSR18] uses convolutional layers (g is a non-linear activation function, $*$ is the linear convolution operator, vec indicates 2D reshaping):

$$f_{ConvE} = \langle \sigma\left(vec\left(g\left([\overline{\mathbf{e}_s}; \overline{\mathbf{r}_p}] * \Omega\right)\right)\mathbf{W}\right)\mathbf{e}_o \rangle$$

- *ConvKB* [NNNP18] uses convolutional layers and a dot product:

$$f_{ConvKB} = concat\left(g\left([\mathbf{e}_s, \mathbf{r}_p, \mathbf{e}_o]\right) * \Omega\right)) \cdot W$$

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[, loss_params, verbose])	Pairwise, max-margin loss.
AbsoluteMarginLoss(eta[, loss_params, verbose])	Absolute margin , max-margin loss.
SelfAdversarialLoss(eta[, loss_params, verbose])	Self adversarial sampling loss.
NLLLoss(eta[, loss_params, verbose])	Negative log-likelihood loss.
NLLMulticlass(eta[, loss_params, verbose])	Multiclass NLL Loss.
BCELoss(eta[, loss_params, verbose])	Binary Cross Entropy Loss.

PairwiseLoss

class ampligraph.latent_features.**PairwiseLoss**(*eta*, *loss_params=None*, *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[, loss_params, verbose])	Initialize Loss.

__init__(*eta*, *loss_params=None*, *verbose=False*)

Initialize Loss.

Parameters

- **eta** (*int*) – Number of negatives.
- **loss_params** (*dict*) – Dictionary of loss-specific hyperparams:
  - **'margin'**: (float). Margin to be used in pairwise loss computation (default: 1)

  Example: `loss_params={'margin': 1}`

AbsoluteMarginLoss

class ampligraph.latent_features.**AbsoluteMarginLoss**(*eta*, *loss_params=None*, *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[, loss_params, verbose])	Initialize Loss

__init__(*eta*, *loss_params=None*, *verbose=False*)

Initialize Loss

Parameters

- **eta** (*int*) – Number of negatives.

- **loss_params** (*dict*) – Dictionary of loss-specific hyperparams:
  - **'margin'**: float. Margin to be used in pairwise loss computation (default:1)

  Example: `loss_params={'margin': 1}`

SelfAdversarialLoss

class ampligraph.latent_features.**SelfAdversarialLoss**(*eta*, *loss_params=None*, *verbose=False*)

Self adversarial sampling loss.

Introduced in [SDNT19].

$$\mathcal{L} = -log\,\sigma(\gamma + f_{model}(\mathbf{s}, \mathbf{o})) - \sum_{i=1}^{n} p(h_i^{'}, r, t_i^{'})\; log\;\sigma(-f_{model}(\mathbf{s}_i^{'}, \mathbf{o}_i^{'}) - \gamma)$$

where $\mathbf{s}, \mathbf{o} \in \mathcal{R}^k$ are the embeddings of the subject and object of a triple $t = (s, r, o)$, γ is the margin, σ the sigmoid function, and $p(s_i^{'}, r, o_i^{'})$ is the negatives sampling distribution which is defined as:

$$p(s'_j, r, o'_j | \{(s_i, r_i, o_i)\}) = \frac{\exp \alpha\, f_{model}(\mathbf{s'_j}, \mathbf{o'_j})}{\sum_i \exp \alpha\, f_{model}(\mathbf{s'_i}, \mathbf{o'_i})}$$

where α is the temperature of sampling, f_{model} is the scoring function of the desired embeddings model.

Methods

`__init__`(eta[, loss_params, verbose])	Initialize Loss

__init__(*eta*, *loss_params=None*, *verbose=False*)

Initialize Loss

Parameters

- **eta** (*int*) – number of negatives
- **loss_params** (*dict*) – Dictionary of loss-specific hyperparams:
  - **'margin'**: (float). Margin to be used for loss computation (default: 1)
  - **'alpha'** : (float). Temperature of sampling (default:0.5)

  Example: `loss_params={'margin': 1, 'alpha': 0.5}`

NLLLoss

class ampligraph.latent_features.**NLLLoss**(*eta*, *loss_params=None*, *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 $y \in -1, 1$ is the label of the statement, $\mathcal{G}$ 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[, loss_params, verbose])	Initialize Loss.

__init__ (*eta*, *loss_params=None*, *verbose=False*)
Initialize Loss.

Parameters

- **eta** (*int*) – Number of negatives.
- **loss_params** (*dict*) – Dictionary of hyperparams. No hyperparameters are required for this loss.

NLLMulticlass

class ampligraph.latent_features.**NLLMulticlass**(*eta*, *loss_params=None*, *verbose=False*)

Multiclass NLL Loss.

Introduced in [aC15] where both the subject and objects are corrupted (to use it in this way pass corrupt_sides = ['s', 'o'] to embedding_model_params) .

This loss was re-engineered in [KBK17] where only the object was corrupted to get improved performance (to use it in this way pass corrupt_sides = 'o' to embedding_model_params).

$$\mathcal{L}(\mathcal{X}) = - \sum_{x_{e_1,e_2,r_k} \in X} log\, p(e_2|e_1, r_k) - \sum_{x_{e_1,e_2,r_k} \in X} log\, p(e_1|r_k, e_2)$$

Examples

```
>>> from ampligraph.latent_features import TransE
>>> model = TransE(batches_count=1, seed=555, epochs=20, k=10,
>>>                embedding_model_params={'corrupt_sides':['s', 'o']},
>>>                loss='multiclass_nll', loss_params={})
```

Methods

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

__init__ (*eta*, *loss_params=None*, *verbose=False*)
Initialize Loss

Parameters

- **eta** (*int*) – number of negatives
- **loss_params** (*dict*) – Dictionary of loss-specific hyperparams:

BCELoss

class ampligraph.latent_features.**BCELoss**(*eta*, *loss_params={}*, *verbose=False*)

Binary Cross Entropy Loss.

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot log(p(y_i)) + (1 - y_i) \cdot log(1 - p(y_i))$$

Examples

```
>>> from ampligraph.latent_features.models import ConvE
>>> model = ConvE(batches_count=1, seed=555, epochs=20, k=10, loss='bce', loss_
↪params={})
```

Methods

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

__init__(*eta*, *loss_params={}*, *verbose=False*)

Initialize Loss

Parameters **loss_params** (*dict*) – Dictionary of loss-specific hyperparams:

Regularizers

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

LPRegularizer([regularizer_params, verbose])	Performs LP regularization

LPRegularizer

class ampligraph.latent_features.**LPRegularizer**(*regularizer_params=None*, *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, $p \in 1, 2, 3$ is the p-norm and λ is the regularization weight.

For example, if $p = 1$ the function will perform L1 regularization. L2 regularization is obtained with $p = 2$.

The nuclear 3-norm proposed in the ComplEx-N3 paper [LUO18] can be obtained with `regularizer_params={'p': 3}`.

Methods

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

__init__ (*regularizer_params=None*, *verbose=False*)
Initializes the hyperparameters needed by the algorithm.

Parameters **regularizer_params** (*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)

Example: `regularizer_params={'lambda': 1e-5, 'p': 1}`

Initializers

AmpliGraph includes a number of initializers that can be used to initialize the embeddings. They can be passed as hyperparameter, and they can be thus used *during model selection*.

RandomNormal([initializer_params, verbose, seed])	Initializes from a normal distribution with specified `mean` and `std`
RandomUniform([initializer_params, verbose, ...])	Initializes from a uniform distribution with specified `low` and `high`
Xavier([initializer_params, verbose, seed])	Follows the xavier strategy for initialization of layers [GB10].

RandomNormal

class ampligraph.latent_features.**RandomNormal** (*initializer_params={}*, *verbose=True*, *seed=0*)
Initializes from a normal distribution with specified `mean` and `std`

$$\mathcal{N}(\mu, \sigma)$$

Methods

__init__([initializer_params, verbose, seed])	Initialize the Random Normal initialization strategy

__init__ (*initializer_params={}*, *verbose=True*, *seed=0*)
Initialize the Random Normal initialization strategy

Parameters

- **initializer_params** (*dict*) – Consists of key-value pairs. The initializer will check the keys to get the corresponding params:
  - **mean**: (float). Mean of the weights(default: 0)
  - **std**: (float): std of the weights (default: 0.05)

Example: `initializer_params={'mean': 0, 'std': 0.01}`

- **verbose** (*bool*) – Enable/disable verbose mode
- **seed** (*int/np.random.RandomState*) – random state for random number generator

RandomUniform

class ampligraph.latent_features.**RandomUniform**(*initializer_params={}*, *verbose=True*, *seed=0*)

Initializes from a uniform distribution with specified `low` and `high`

$$\mathcal{U}(low, high)$$

Methods

__init__([initializer_params, verbose, seed])	Initialize the Uniform initialization strategy

__init__(*initializer_params={}*, *verbose=True*, *seed=0*)

Initialize the Uniform initialization strategy

Parameters

- **initializer_params** (*dict*) – Consists of key-value pairs. The initializer will check the keys to get the corresponding params:
  - **low**: (float). lower bound for uniform number (default: -0.05)
  - **high**: (float): upper bound for uniform number (default: 0.05)

  Example: `initializer_params={'low': 0, 'high': 0.01}`
- **verbose** (*bool*) – Enable/disable verbose mode
- **seed** (*int/np.random.RandomState*) – random state for random number generator

Xavier

class ampligraph.latent_features.**Xavier**(*initializer_params={}*, *verbose=True*, *seed=0*)

Follows the xavier strategy for initialization of layers [GB10].

If `uniform` is set to True, then it initializes the layer from the following uniform distribution:

$$\mathcal{U}(-\sqrt{\frac{6}{fan_{in} + fan_{out}}}, \sqrt{\frac{6}{fan_{in} + fan_{out}}})$$

If `uniform` is False, then it initializes the layer from the following normal distribution:

$$\mathcal{N}(0, \sqrt{\frac{2}{fan_{in} + fan_{out}}})$$

where fan_{in} and fan_{out} are number of input units and output units of the layer respectively.

Methods

__init__([initializer_params, verbose, seed])	Initialize the Xavier strategy

__init__ (*initializer_params={}*, *verbose=True*, *seed=0*)
Initialize the Xavier strategy

Parameters

- **initializer_params** (*dict*) – Consists of key-value pairs. The initializer will check the keys to get the corresponding params:
  - **uniform**: (bool). indicates whether to use Xavier Uniform or Xavier Normal initializer.

  Example: `initializer_params={'uniform': False}`
- **verbose** (*bool*) – Enable/disable verbose mode
- **seed** (*int/np.random.RandomState*) – random state for random number generator

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.

Saving/Restoring Models

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

More details in the *utils* module.

3.3.3 Evaluation

The module includes 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 *ampligraph.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) *ampligraph.evaluation.mr_score()*. 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 or list, shape [n] or [n,2]*) – Input ranks of n test statements.

Returns **mrr_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 can be used in conjunction with the learning to rank evaluation protocol of *ampligraph.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 presented along the more reliable MRR *ampligraph.evaluation.mrr_score()*.

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
```

Parameters **ranks** (*ndarray or list, shape [n] or [n,2]*) – Input ranks of n test statements.

Returns **mr_score** – The MR score

Return type float

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 can be used in conjunction with the learning to rank evaluation protocol of *ampligraph.evaluation.evaluate_performance()*.

It is formally defined as follows:

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

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
```

(continues on next page)

(continued from previous page)

```
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 or list, shape [n] or [n,2]*) – Input ranks of n test 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[, ...])	Generate corruptions for training.

generate_corruptions_for_eval

ampligraph.evaluation.**generate_corruptions_for_eval**(*X*, *entities_for_corruption*, *corrupt_side='s, o'*)

Generate corruptions for evaluation.

Create corruptions (subject and object) for a given triple x, in compliance with the local closed world assumption (LCWA), as described in [NMTG16].

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 of the triple to corrupt:
  - 's': corrupt only subject.
  - 'o': corrupt only object
  - 's+o': corrupt both subject and object
  - 's,o': corrupt both subject and object but ranks are computed separately.

Returns **out** – An array of corruptions for the triples for x.

Return type Tensor, shape [n, 3]

generate_corruptions_for_fit

ampligraph.evaluation.**generate_corruptions_for_fit**(*X*, *entities_list=None*, *eta=1*, *corrupt_side='s*, *o'*, *entities_size=0*, *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).

Note: When processing large knowledge graphs, it may be useful to generate corruptions only using entities from a single batch. This also brings the benefit of creating more meaningful negatives, as entities used to corrupt are sourced locally. The function can be configured to generate corruptions *only* using the entities from the current batch. You can enable such behaviour be setting `entities_size=0`. In such case, if `entities_list=None` all entities from the *current batch* will be used to generate corruptions.

Parameters

- **X** (*Tensor, shape [n, 3]*) – An array of positive triples that will be used to create corruptions.
- **entities_list** (*list*) – List of entities to be used for generating corruptions. (default:None).

  If `entities_list=None` and `entities_size` is the number of all entities, all entities will be used to generate corruptions (default behaviour).

  If `entities_list=None` and `entities_size=0`, the batch entities will be used to generate corruptions.
- **eta** (*int*) – The number of corruptions per triple that must be generated.
- **corrupt_side** (*string*) – Specifies which side of the triple to corrupt:
  - 's': corrupt only subject.
  - 'o': corrupt only object
  - 's+o': corrupt both subject and object
  - 's,o': corrupt both subject and object

- **entities_size** (*int*) – Size of entities to be used while generating corruptions. It assumes entity id's start from 0 and are continuous. (default: 0). When processing large knowledge graphs, it may be useful to generate corruptions only using entities from a single batch. This also brings the benefit of creating more meaningful negatives, as entities used to corrupt are sourced locally. The function can be configured to generate corruptions *only* using the entities from the current batch. You can enable such behaviour be setting `entities_size=0`. In such case, if `entities_list=None` all entities from the *current batch* will be used to generate corruptions.
- **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, ...)	Model selection routine for embedding models via either grid search or random search.

evaluate_performance

ampligraph.evaluation.**evaluate_performance**(*X*, *model*, *filter_triples=None*, *verbose=False*, *filter_unseen=True*, *entities_subset=None*, *corrupt_side='s, o'*, *use_default_protocol=False*)

Evaluate the performance of an embedding model.

The evaluation protocol follows the procedure defined in [BUGD+13] and can be summarised as:

1. Artificially generate negative triples by corrupting first the subject and then the object.
2. Remove the positive triples from the set returned by (1) – positive triples are usually the concatenation of training, validation and test sets.
3. Rank each test triple against all remaining triples returned by (2).

With the ranks of both object and subject corruptions, one may compute metrics such as the MRR by calculating them separately and then averaging them out. Note that the metrics implemented in AmpliGraph's `evaluate.metrics` module will already work that way when provided with the input returned by `evaluate_performance`.

The artificially generated negatives are compliant with the local closed world assumption (LCWA), as described in [NMTG16]. In practice, that means only one side of the triple is corrupted at a time (i.e. either the subject or the object).

Note: The evaluation protocol assigns the worst rank to a positive test triple in case of a tie with negatives. This is the agreed upon behaviour in literature.

Hint: When `entities_subset=None`, the method will use all distinct entities in the knowledge graph `X` to generate negatives to rank against. This might slow down the eval. Some of the corruptions may not even make sense for the task that one may be interested in.

For eg, consider the case <Actor, acted_in, ?>, where we are mainly interested in such movies that an actor has acted in. A sensible way to evaluate this would be to rank against all the movie entities and compute the desired metrics. In such cases, where focus us on particular task, it is recommended to pass the desired entities to use to generate corruptions to `entities_subset`. Besides, trying to rank a positive against an extremely large number of negatives may be overkilling.

As a reference, the popular FB15k-237 dataset has ~15k distinct entities. The evaluation protocol ranks each positives against 15k corruptions per side.

Parameters

- **X** (*ndarray, shape [n, 3]*) – An array of test triples.
- **model** (EmbeddingModel) – A knowledge graph embedding model
- **filter_triples** (*ndarray of shape [n, 3] or None*) – The triples used to filter negatives.

  Note: When *filtered* mode is enabled (i.e. *filtered_triples* is not `None`), to speed up the procedure, we use a database based filtering. This strategy is as described below:

  - Store the filter_triples in the DB
  - For each test triple, we generate corruptions for evaluation and score them.
  - The corruptions may contain some False Negatives. We find such statements by quering the database.
  - From the computed scores we retrieve the scores of the False Negatives.
  - We compute the rank of the test triple by comparing against ALL the corruptions.
  - We then compute the number of False negatives that are ranked higher than the test triple; and then subtract this value from the above computed rank to yield the final filtered rank.

  Execution Time: This method takes ~4 minutes on FB15K using ComplEx (Intel Xeon Gold 6142, 64 GB Ubuntu 16.04 box, Tesla V100 16GB)

- **verbose** (*bool*) – Verbose mode
- **filter_unseen** (*bool*) – This can be set to False to skip filtering of unseen entities if train_test_split_unseen() was used to split the original dataset.
- **entities_subset** (*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 of the triple to corrupt:
  - 's': corrupt only subject.
  - 'o': corrupt only object.
  - 's+o': corrupt both subject and object.
  - 's,o': corrupt subject and object sides independently and return 2 ranks. This corresponds to the evaluation protocol used in literature, where head and tail corruptions are evaluated separately.

  Note: When `corrupt_side='s,o'` the function will return 2*n ranks as a [n, 2] array. The first column of the array represents the subject corruptions. The second

column of the array represents the object corruptions. Otherwise, the function returns n ranks as [n] array.

- **use_default_protocol** (*bool*) – Flag to indicate whether to use the standard protocol used in literature defined in [BUGD+13] (default: False). If set to *True*, `corrupt_side` will be set to *'s,o'*. This corresponds to the evaluation protocol used in literature, where head and tail corruptions are evaluated separately, i.e. in corrupt_side='s,o' mode

Returns **ranks** – An array of ranks of test triples. When `corrupt_side='s,o'` the function returns [n,2]. The first column represents the rank against subject corruptions and the second column represents the rank against object corruptions. In other cases, it returns [n] i.e. rank against the specified corruptions.

Return type ndarray, shape [n] or [n,2] depending on the value of corrupt_side.

Examples

```
>>> 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
>>>
>>> X = load_wn18()
>>> model = ComplEx(batches_count=10, seed=0, epochs=10, k=150, eta=1,
>>>                 loss='nll', optimizer='adam')
>>> model.fit(np.concatenate((X['train'], X['valid'])))
>>>
>>> filter_triples = np.concatenate((X['train'], X['valid'], X['test']))
>>> ranks = evaluate_performance(X['test'][:5], model=model,
>>>                              filter_triples=filter_triples,
>>>                              corrupt_side='s+o',
>>>                              use_default_protocol=False)
>>> ranks
array([  1, 582, 543,   6,  31])
>>> mrr_score(ranks)
0.24049691297347323
>>> hits_at_n_score(ranks, n=10)
0.4
```

select_best_model_ranking

ampligraph.evaluation.**select_best_model_ranking**(*model_class*, *X_train*, *X_valid*, *X_test*, *param_grid*, *max_combinations=None*, *param_grid_random_seed=0*, *use_filter=True*, *early_stopping=False*, *early_stopping_params=None*, *use_test_for_selection=False*, *entities_subset=None*, *corrupt_side='s, o'*, *use_default_protocol=False*, *retrain_best_model=False*, *verbose=False*)

Model selection routine for embedding models via either grid search or random search.

For grid search, pass a fixed `param_grid` and leave `max_combinations` as *None* so that all combinations will be explored.

For random search, delimit `max_combinations` to your computational budget and optionally set some parameters to be callables instead of a list (see the documentation for `param_grid`).

Note: Random search is more efficient than grid search as the number of parameters grows [BB12]. It is also a strong baseline against more advanced methods such as Bayesian optimization [LJ18].

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

Note we generate negatives at runtime according to the strategy described in [BUGD+13].

Note: By default, model selection is done with raw MRR for better runtime performance (`use_filter=False`).

Parameters

- **model_class** (*class*) – The class of the EmbeddingModel to evaluate (TransE, DistMult, ComplEx, etc).
- **X_train** (*ndarray, shape [n, 3]*) – An array of training triples.
- **X_valid** (*ndarray, shape [n, 3]*) – An array of validation triples.
- **X_test** (*ndarray, shape [n, 3]*) – An array of test triples.
- **param_grid** (*dict*) – A grid of hyperparameters to use in model selection. The routine will train a model for each combination of these hyperparameters.

  Parameters can be either callables or lists. If callable, it must take no parameters and return a constant value. If any parameter is a callable, `max_combinations` must be set to some value.

  For example, the learning rate could either be `"lr": [0.1, 0.01]` or `"lr": lambda: np.random.uniform(0.01, 0.1)`.
- **max_combinations** (*int*) – Maximum number of combinations to explore. By default (None) all combinations will be explored, which makes it incompatible with random parameters for random search.

- **param_grid_random_seed** (*int*) – Random seed for the parameters that are callables and random.
- **use_filter** (*bool*) – If True, will use the entire input dataset X to compute filtered MRR (default: True).
- **early_stopping** (*bool*) – Flag to enable early stopping (default:False).

  If set to `True`, the training loop adopts the following early stopping heuristic:

  - The model will be trained regardless of early stopping for `burn_in` epochs.
  - Every `check_interval` epochs the method will compute the metric specified in `criteria`.

  If such metric decreases for `stop_interval` checks, we stop training early.

  Note the metric is computed on `x_valid`. This is usually a validation set that you held out.

  Also, because `criteria` is a ranking metric, it requires generating negatives. Entities used to generate corruptions can be specified, as long as the side(s) of a triple to corrupt. The method supports filtered metrics, by passing an array of positives to `x_filter`. This will be used to filter the negatives generated on the fly (i.e. the corruptions).

  Note: Keep in mind the early stopping criteria may introduce a certain overhead (caused by the metric computation). The goal is to strike a good trade-off between such overhead and saving training epochs.

  A common approach is to use MRR unfiltered:

  ```
  early_stopping_params={x_valid=X['valid'], 'criteria': 'mrr'}
  ```

  Note the size of validation set also contributes to such overhead. In most cases a smaller validation set would be enough.

- **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: False).
- **entities_subset** (*array-like*) – List of entities to use for corruptions. If None, will generate corruptions using all distinct entities (default: 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. `s,o` is to corrupt both subject and object but ranks are computed separately (default).

- **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 triples i.e. corrupt_side='s,o' mode.
- **retrain_best_model** (*bool*) – Flag to indicate whether best model should be re-trained at the end with the validation set used in the search. Default: False.
- **verbose** (*bool*) – Verbose mode for the model selection procedure (which is independent of the verbose mode in the model fit).

  Verbose mode includes display of the progress bar, logging info for each iteration, evaluation information, and exception details.

  If you need verbosity inside the model training itself, change the verbose parameter within the `param_grid`.

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] or [n,2] depending on the value of corrupt_side.*) – An array of ranks of test triples. When `corrupt_side='s,o'` the function returns [n,2]. The first column represents the rank against subject corruptions and the second column represents the rank against object corruptions. In other cases, it returns [n] i.e. rank against the specified corruptions.
- **mrr_test** (*float*) – The MRR (filtered) of the best model, retrained on the concatenation of training and validation sets, computed over the test set.
- **experimental_history** (*list of dict*) – A list containing all the intermediate experimental results: the model parameters and the corresponding validation metrics.

Examples

```
>>> from ampligraph.datasets import load_wn18
>>> from ampligraph.latent_features import ComplEx
>>> from ampligraph.evaluation import select_best_model_ranking
>>> import numpy as np
>>>
>>> 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": {
```

(continues on next page)

(continued from previous page)

```
>>>                         "margin": [2]
>>>                     },
>>>                     "embedding_model_params": {
>>>
>>>                     },
>>>                     "regularizer": ["LP", None],
>>>                     "regularizer_params": {
>>>                         "p": [1, 3],
>>>                         "lambda": [1e-4, 1e-5]
>>>                     },
>>>                     "optimizer": ["adagrad", "adam"],
>>>                     "optimizer_params":{
>>>                         "lr": lambda: np.random.uniform(0.0001, 0.01)
>>>                     },
>>>                     "verbose": False
>>>                 }
>>> select_best_model_ranking(model_class, X['train'], X['valid'], X['test'],␣
↪param_grid,
>>>                           max_combinations=100, 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, ...])	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=100*, *seed=0*, *allow_duplication=False*)

Split into train and test sets.

This function carves out a test set that 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.
- **allow_duplication** (*boolean*) – Flag to indicate if the test set can contain duplicated triples.

Returns

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

Examples

```
>>> import numpy as np
>>> from ampligraph.evaluation import train_test_split_no_unseen
>>> # load your dataset to X
>>> X = np.array([['a', 'y', 'b'],
>>>               ['f', 'y', 'e'],
>>>               ['b', 'y', 'a'],
>>>               ['a', 'y', 'c'],
>>>               ['c', 'y', 'a'],
>>>               ['a', 'y', 'd'],
>>>               ['c', 'y', 'd'],
>>>               ['b', 'y', 'c'],
>>>               ['f', 'y', 'e']])
>>> # if you want to split into train/test datasets
>>> X_train, X_test = train_test_split_no_unseen(X, test_size=2)
>>> X_train
array([['a', 'y', 'b'],
    ['f', 'y', 'e'],
    ['b', 'y', 'a'],
    ['c', 'y', 'a'],
    ['c', 'y', 'd'],
    ['b', 'y', 'c'],
    ['f', 'y', 'e']], dtype='<U1')
>>> X_test
array([['a', 'y', 'c'],
    ['a', 'y', 'd']], dtype='<U1')
>>> # if you want to split into train/valid/test datasets, call it 2 times
>>> X_train_valid, X_test = train_test_split_no_unseen(X, test_size=2)
>>> X_train, X_valid = train_test_split_no_unseen(X_train_valid, test_size=2)
>>> X_train
array([['a', 'y', 'b'],
    ['b', 'y', 'a'],
    ['c', 'y', 'd'],
    ['b', 'y', 'c'],
    ['f', 'y', 'e']], dtype='<U1')
>>> X_valid
array([['f', 'y', 'e'],
    ['c', 'y', 'a']], dtype='<U1')
>>> X_test
array([['a', 'y', 'c'],
    ['a', 'y', 'd']], dtype='<U1')
```

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]

3.3.4 Discovery

This module includes a number of functions to perform knowledge discovery in graph embeddings.

Functions provided include `discover_facts` which will generate candidate statements using one of several defined strategies and return triples that perform well when evaluated against corruptions, `find_clusters` which will perform link-based cluster analysis on a knowledge graph, `find_duplicates` which will find duplicate entities in a graph based on their embeddings, and `query_topn` which when given two elements of a triple will return the top_n results of all possible completions ordered by predicted score.

discover_facts(X, model[, top_n, strategy, ...])	Discover new facts from an existing knowledge graph.
find_clusters(X, model[, ...])	Perform link-based cluster analysis on a knowledge graph.
find_duplicates(X, model[, mode, metric, ...])	Find duplicate entities, relations or triples in a graph based on their embeddings.
query_topn(model[, top_n, head, relation, ...])	Queries the model with two elements of a triple and returns the top_n results of all possible completions ordered by score predicted by the model.

discover_facts

ampligraph.discovery.**discover_facts**(*X*, *model*, *top_n=10*, *strategy='random_uniform'*, *max_candidates=100*, *target_rel=None*, *seed=0*)

Discover new facts from an existing knowledge graph.

You should use this function when you already have a model trained on a knowledge graph and you want to discover potentially true statements in that knowledge graph.

The general procedure of this function is to generate a set of candidate statements C according to some sampling strategy `strategy`, then rank them against a set of corruptions using the *ampligraph.evaluation.evaluate_performance()* function. Candidates that appear in the `top_n` ranked statements of this procedure are returned as likely true statements.

The majority of the strategies are implemented with the same underlying principle of searching for candidate statements:

- from among the less frequent entities ('entity_frequency'),
- less connected entities ('graph_degree', cluster_coefficient'),
- less frequent local graph structures ('cluster_triangles', 'cluster_squares'), on the assumption that densely connected entities are less likely to have missing true statements.

- The remaining strategies ('random_uniform', 'exhaustive') generate candidate statements by a random sampling of entity and relations and exhaustively, respectively.

Warning: Due to the significant amount of computation required to evaluate all triples using the 'exhaustive' strategy, we do not recommend its use at this time.

The function will automatically filter entities that haven't been seen by the model, and operates on the assumption that the model provided has been fit on the data `X` (determined heuristically), although `X` may be a subset of the original data, in which case a warning is shown.

The `target_rel` argument indicates what relation to generate candidate statements for. If this is set to `None` then all target relations will be considered for sampling.

Parameters

- **X** (*ndarray, shape [n, 3]*) – The input knowledge graph used to train `model`, or a subset of it.
- **model** (`EmbeddingModel`) – The trained model that will be used to score candidate facts.
- **top_n** (*int*) – The cutoff position in ranking to consider a candidate triple as true positive.
- **strategy** (*string*) – The candidates generation strategy:
  - 'random_uniform' : generates N candidates (N <= max_candidates) based on a uniform sampling of entities.
  - 'entity_frequency' : generates candidates by weighted sampling of entities using entity frequency.
  - 'graph_degree' : generates candidates by weighted sampling of entities with graph degree.
  - 'cluster_coefficient' : generates candidates by weighted sampling entities with clustering coefficient.
  - 'cluster_triangles' : generates candidates by weighted sampling entities with cluster triangles.
  - 'cluster_squares' : generates candidates by weighted sampling entities with cluster squares.
- **max_candidates** (*int or float*) – The maximum numbers of candidates generated by 'strategy'. Can be an absolute number or a percentage [0,1] of the size of the `` `X` `` parameter.
- **target_rel** (*str or list(str)*) – Target relations to focus on. The function will discover facts only for that specific relation types. If None, the function attempts to discover new facts for all relation types in the graph.
- **seed** (*int*) – Seed to use for reproducible results.

Returns **X_pred** – A list of new facts predicted to be true.

Return type ndarray, shape [n, 3]

Examples

```
>>> import requests
>>> from ampligraph.datasets import load_from_csv
>>> from ampligraph.latent_features import ComplEx
>>> from ampligraph.discovery import discover_facts
>>>
>>> # Game of Thrones relations dataset
>>> url = 'https://ampligraph.s3-eu-west-1.amazonaws.com/datasets/GoT.csv'
>>> open('GoT.csv', 'wb').write(requests.get(url).content)
>>> X = load_from_csv('.', 'GoT.csv', sep=',')
>>>
>>> model = ComplEx(batches_count=10, seed=0, epochs=200, k=150, eta=5,
>>>                 optimizer='adam', optimizer_params={'lr':1e-3},
>>>                 loss='multiclass_nll', regularizer='LP',
>>>                 regularizer_params={'p':3, 'lambda':1e-5},
>>>                 verbose=True)
>>> model.fit(X)
>>>
>>> discover_facts(X, model, top_n=3, max_candidates=20000, strategy='entity_
↪frequency',
>>>                target_rel='ALLIED_WITH', seed=42)
array([['House Reed of Greywater Watch', 'ALLIED_WITH', 'Sybelle Glover'],
       ['Hugo Wull', 'ALLIED_WITH', 'House Norrey'],
       ['House Grell', 'ALLIED_WITH', 'Delonne Allyrion'],
       ['Lorent Lorch', 'ALLIED_WITH', 'House Ruttiger']], dtype=object)
```

find_clusters

ampligraph.discovery.**find_clusters**(*X*, *model*, *clustering_algorithm=DBSCAN(algorithm='auto', eps=0.5, leaf_size=30, metric='euclidean', metric_params=None, min_samples=5, n_jobs=None, p=None)*, *mode='entity'*)

Perform link-based cluster analysis on a knowledge graph.

The clustering happens on the embedding space of the entities and relations. For example, if we cluster some entities of a model that uses *k=100* (i.e. embedding space of size 100), we will apply the chosen clustering algorithm on the 100-dimensional space of the provided input samples.

Clustering can be used to evaluate the quality of the knowledge embeddings, by comparing to natural clusters. For example, in the example below we cluster the embeddings of international football matches and end up finding geographical clusters very similar to the continents. This comparison can be subjective by inspecting a 2D projection of the embedding space or objective using a clustering metric.

The choice of the clustering algorithm and its corresponding tuning will greatly impact the results. Please see scikit-learn documentation for a list of algorithms, their parameters, and pros and cons.

Clustering is exclusive (i.e. a triple is assigned to one and only one cluster).

Parameters

- **X** (*ndarray, shape [n, 3] or [n]*) – The input to be clustered. `X` can either be the triples of a knowledge graph, its entities, or its relations. The argument `mode` defines whether `X` is supposed an array of triples or an array of either entities or relations.

- **model** (EmbeddingModel) – The fitted model that will be used to generate the embeddings. This model must have been fully trained already, be it directly with `fit()` or from a helper function such as *ampligraph.evaluation.select_best_model_ranking()*.
- **clustering_algorithm** (*object*) – The initialized object of the clustering algorithm. It should be ready to apply the *fit_predict* method. Please see: scikit-learn documentation to understand the clustering API provided by scikit-learn. The default clustering model is sklearn's DBSCAN with its default parameters.
- **mode** (*string*) – Clustering mode. Choose from:
  - 'entity' (default): the algorithm will cluster the embeddings of the provided entities.
  - 'relation': the algorithm will cluster the embeddings of the provided relations.
  - 'triple' : the algorithm will cluster the concatenation of the embeddings of the subject, predicate and object for each triple.

Returns **labels** – Index of the cluster each triple belongs to.

Return type ndarray, shape [n]

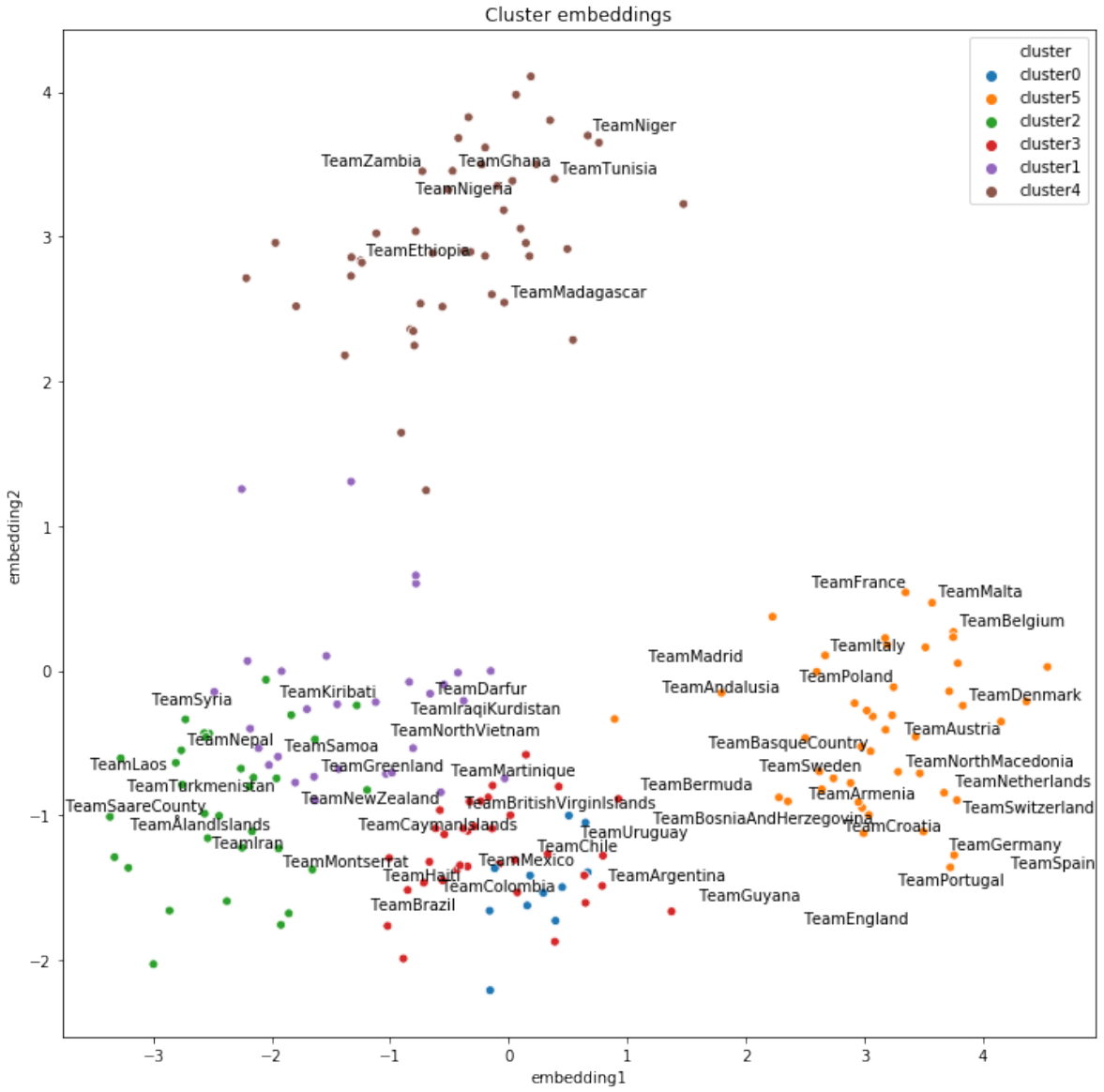
Examples

```
>>> # Note seaborn, matplotlib, adjustText are not AmpliGraph dependencies.
>>> # and must therefore be installed manually as:
>>> #
>>> # $ pip install seaborn matplotlib adjustText
>>>
>>> import requests
>>> import pandas as pd
>>> import numpy as np
>>> from sklearn.decomposition import PCA
>>> from sklearn.cluster import KMeans
>>> import matplotlib.pyplot as plt
>>> import seaborn as sns
>>>
>>> # adjustText lib: https://github.com/Phlya/adjustText
>>> from adjustText import adjust_text
>>>
>>> from ampligraph.datasets import load_from_csv
>>> from ampligraph.latent_features import ComplEx
>>> from ampligraph.discovery import find_clusters
>>>
>>> # International football matches triples
>>> # See tutorial here to understand how the triples are created from a tabular␣
↪dataset:
>>> # https://github.com/Accenture/AmpliGraph/blob/master/docs/tutorials/
↪ClusteringAndClassificationWithEmbeddings.ipynb
>>> url = 'https://ampligraph.s3-eu-west-1.amazonaws.com/datasets/football.csv'
>>> open('football.csv', 'wb').write(requests.get(url).content)
>>> X = load_from_csv('.', 'football.csv', sep=',')[:, 1:]
>>>
>>> model = ComplEx(batches_count=50,
>>>                 epochs=300,
>>>                 k=100,
```

(continues on next page)

(continued from previous page)

```
>>>                 eta=20,
>>>                 optimizer='adam',
>>>                 optimizer_params={'lr':1e-4},
>>>                 loss='multiclass_nll',
>>>                 regularizer='LP',
>>>                 regularizer_params={'p':3, 'lambda':1e-5},
>>>                 seed=0,
>>>                 verbose=True)
>>> model.fit(X)
>>>
>>> df = pd.DataFrame(X, columns=["s", "p", "o"])
>>>
>>> teams = np.unique(np.concatenate((df.s[df.s.str.startswith("Team")],
>>>                                   df.o[df.o.str.startswith("Team")])))
>>> team_embeddings = model.get_embeddings(teams, embedding_type='entity')
>>>
>>> embeddings_2d = PCA(n_components=2).fit_transform(np.array([i for i in team_
↪embeddings]))
>>>
>>> # Find clusters of embeddings using KMeans
>>> kmeans = KMeans(n_clusters=6, n_init=100, max_iter=500)
>>> clusters = find_clusters(teams, model, kmeans, mode='entity')
>>>
>>> # Plot results
>>> df = pd.DataFrame({"teams": teams, "clusters": "cluster" + pd.
↪Series(clusters).astype(str),
>>>                    "embedding1": embeddings_2d[:, 0], "embedding2":␣
↪embeddings_2d[:, 1]})
>>>
>>> plt.figure(figsize=(10, 10))
>>> plt.title("Cluster embeddings")
>>>
>>> ax = sns.scatterplot(data=df, x="embedding1", y="embedding2", hue="clusters")
>>>
>>> texts = []
>>> for i, point in df.iterrows():
>>>     if np.random.uniform() < 0.1:
>>>         texts.append(plt.text(point['embedding1']+.02, point['embedding2'],␣
↪str(point['teams'])))
>>> adjust_text(texts)
```



find_duplicates

ampligraph.discovery.**find_duplicates**(*X*, *model*, *mode='entity'*, *metric='l2'*, *tolerance='auto'*, *expected_fraction_duplicates=0.1*, *verbose=False*)

Find duplicate entities, relations or triples in a graph based on their embeddings.

For example, say you have a movie dataset that was scraped off the web with possible duplicate movies. The movies in this case are the entities. Therefore, you would use the 'entity' mode to find all the movies that could de duplicates of each other.

Duplicates are defined as points whose distance in the embedding space are smaller than some given threshold (called the tolerance).

The tolerance can be defined a priori or be found via an optimisation procedure given an expected fraction of duplicates. The optimisation algorithm applies a root-finding routine to find the tolerance that gets to the closest expected fraction. The routine always converges.

Distance is defined by the chosen metric, which by default is the Euclidean distance (L2 norm).

As the distances are calculated on the embedding space, the embeddings must be meaningful for this routine to work properly. Therefore, it is suggested to evaluate the embeddings first using a metric such as MRR before considering applying this method.

Parameters

- **X** (*ndarray, shape [n, 3] or [n]*) – The input to be clustered. X can either be the triples of a knowledge graph, its entities, or its relations. The argument *mode* defines whether X is supposed an array of triples or an array of either entities or relations.
- **model** (EmbeddingModel) – The fitted model that will be used to generate the embeddings. This model must have been fully trained already, be it directly with `fit()` or from a helper function such as *ampligraph.evaluation.select_best_model_ranking()*.
- **mode** (*string*) – Choose from:
  - 'entity' (default): the algorithm will find duplicates of the provided entities based on their embeddings.
  - 'relation': the algorithm will find duplicates of the provided relations based on their embeddings.
  - 'triple' : the algorithm will find duplicates of the concatenation of the embeddings of the subject, predicate and object for each provided triple.
- **metric** (*str*) – A distance metric used to compare entity distance in the embedding space. See options here.
- **tolerance** (*int or str*) – Minimum distance (depending on the chosen `metric`) to define one entity as the duplicate of another. If 'auto', it will be determined automatically in a way that you get the `expected_fraction_duplicates`. The 'auto' option can be much slower than the regular one, as the finding duplicate internal procedure will be repeated multiple times.
- **expected_fraction_duplicates** (*float*) – Expected fraction of duplicates to be found. It is used only when `tolerance` is 'auto'. Should be between 0 and 1 (default: 0.1).
- **verbose** (*bool*) – Whether to print evaluation messages during optimisation (if `tolerance` is 'auto'). Default: False.

Returns

- **duplicates** (*set of frozensets*) – Each entry in the duplicates set is a frozenset containing all entities that were found to be duplicates according to the metric and tolerance. Each frozenset will contain at least two entities.
- **tolerance** (*float*) – Tolerance used to find the duplicates (useful in the case of the automatic tolerance option).

Examples

```
>>> import pandas as pd
>>> import numpy as np
>>> import re
>>>
>>> # The IMDB dataset used here is part of the Movies5 dataset found on:
>>> # The Magellan Data Repository (https://sites.google.com/site/anhaidgroup/
↪projects/data)
>>> import requests
>>> url = 'http://pages.cs.wisc.edu/~anhai/data/784_data/movies5.tar.gz'
>>> open('movies5.tar.gz', 'wb').write(requests.get(url).content)
>>> import tarfile
>>> tar = tarfile.open('movies5.tar.gz', "r:gz")
>>> tar.extractall()
>>> tar.close()
>>>
>>> # Reading tabular dataset of IMDB movies and filling the missing values
>>> imdb = pd.read_csv("movies5/csv_files/imdb.csv")
>>> imdb["directors"] = imdb["directors"].fillna("UnknownDirector")
>>> imdb["actors"] = imdb["actors"].fillna("UnknownActor")
>>> imdb["genre"] = imdb["genre"].fillna("UnknownGenre")
>>> imdb["duration"] = imdb["duration"].fillna("0")
>>>
>>> # Creating knowledge graph triples from tabular dataset
>>> imdb_triples = []
>>>
>>> for _, row in imdb.iterrows():
>>>     movie_id = "ID" + str(row["id"])
>>>     directors = row["directors"].split(",")
>>>     actors = row["actors"].split(",")
>>>     genres = row["genre"].split(",")
>>>     duration = "Duration" + str(int(re.sub("\D", "", row["duration"])) // 30)
>>>
>>>     directors_triples = [(movie_id, "hasDirector", d) for d in directors]
>>>     actors_triples = [(movie_id, "hasActor", a) for a in actors]
>>>     genres_triples = [(movie_id, "hasGenre", g) for g in genres]
>>>     duration_triple = (movie_id, "hasDuration", duration)
>>>
>>>     imdb_triples.extend(directors_triples)
>>>     imdb_triples.extend(actors_triples)
>>>     imdb_triples.extend(genres_triples)
>>>     imdb_triples.append(duration_triple)
>>>
>>> # Training knowledge graph embedding with ComplEx model
>>> from ampligraph.latent_features import ComplEx
>>>
>>> model = ComplEx(batches_count=10,
>>>                 seed=0,
>>>                 epochs=200,
>>>                 k=150,
>>>                 eta=5,
>>>                 optimizer='adam',
>>>                 optimizer_params={'lr':1e-3},
>>>                 loss='multiclass_nll',
>>>                 regularizer='LP',
>>>                 regularizer_params={'p':3, 'lambda':1e-5},
```

(continues on next page)

(continued from previous page)

```
>>>                 verbose=True)
>>>
>>> imdb_triples = np.array(imdb_triples)
>>> model.fit(imdb_triples)
>>>
>>> # Finding duplicates movies (entities)
>>> from ampligraph.discovery import find_duplicates
>>>
>>> entities = np.unique(imdb_triples[:, 0])
>>> dups, _ = find_duplicates(entities, model, mode='entity', tolerance=0.4)
>>> print(list(dups)[:3])
[frozenset({'ID4048', 'ID4049'}), frozenset({'ID5994', 'ID5993'}), frozenset({
↪'ID6447', 'ID6448'})]
>>> print(imdb[imdb.id.isin((4048, 4049, 5994, 5993, 6447, 6448))][['movie_name',
↪'year']])
                    movie_name  year
4048          Ulterior Motives  1993
4049          Ulterior Motives  1993
5993          Chinese Hercules  1973
5994          Chinese Hercules  1973
6447  The Stranglers of Bombay  1959
6448  The Stranglers of Bombay  1959
```

query_topn

ampligraph.discovery.**query_topn**(*model*, *top_n=10*, *head=None*, *relation=None*, *tail=None*, *ents_to_consider=None*, *rels_to_consider=None*)

Queries the model with two elements of a triple and returns the top_n results of all possible completions ordered by score predicted by the model.

For example, given a <subject, predicate> pair in the arguments, the model will score all possible triples <subject, predicate, ?>, filling in the missing element with known entities, and return the top_n triples ordered by score. If given a <subject, object> pair it will fill in the missing element with known relations.

Note: This function does not filter out true statements - triples returned can include those the model was trained on.

Parameters

- **model** (EmbeddingModel) – The trained model that will be used to score triple completions.
- **top_n** (*int*) – The number of completed triples to returned.
- **head** (*string*) – An entity string to query.
- **relation** (*string*) – A relation string to query.
- **tail** – An object string to query.
- **ents_to_consider** (*array-like*) – List of entities to use for triple completions. If None, will generate completions using all distinct entities. (Default: None.)
- **rels_to_consider** (*array-like*) – List of relations to use for triple completions. If None, will generate completions using all distinct relations. (Default: None.)

Returns

- **X** (*ndarray, shape [n, 3]*) – A list of triples ordered by score.
- **S** (*ndarray, shape [n]*) – A list of scores.

Examples

```
>>> import requests
>>> from ampligraph.datasets import load_from_csv
>>> from ampligraph.latent_features import ComplEx
>>> from ampligraph.discovery import discover_facts
>>> from ampligraph.discovery import query_topn
>>>
>>> # Game of Thrones relations dataset
>>> url = 'https://ampligraph.s3-eu-west-1.amazonaws.com/datasets/GoT.csv'
>>> open('GoT.csv', 'wb').write(requests.get(url).content)
>>> X = load_from_csv('.', 'GoT.csv', sep=',')
>>>
>>> model = ComplEx(batches_count=10, seed=0, epochs=200, k=150, eta=5,
>>>                 optimizer='adam', optimizer_params={'lr':1e-3}, loss=
↪'multiclass_nll',
>>>                 regularizer='LP', regularizer_params={'p':3, 'lambda':1e-5},
>>>                 verbose=True)
>>> model.fit(X)
>>>
>>> query_topn(model, top_n=5,
>>>            head='Catelyn Stark', relation='ALLIED_WITH', tail=None,
>>>            ents_to_consider=None, rels_to_consider=None)
>>>
(array([['Catelyn Stark', 'ALLIED_WITH', 'House Tully of Riverrun'],
        ['Catelyn Stark', 'ALLIED_WITH', 'House Stark of Winterfell'],
        ['Catelyn Stark', 'ALLIED_WITH', 'House Wayn'],
        ['Catelyn Stark', 'ALLIED_WITH', 'House Mollen'],
        ['Catelyn Stark', 'ALLIED_WITH', 'Orton Merryweather']],
       dtype='<U44'), array([[10.261374 ],
        [ 8.84298  ],
        [ 2.78139  ],
        [ 1.9809164],
        [ 1.833096 ]], dtype=float32))
```

3.3.5 Utils

This module contains utility functions for neural knowledge graph embedding models.

Saving/Restoring Models

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

save_model(model[, model_name_path])	Save a trained model to disk.
restore_model([model_name_path])	Restore a saved model from disk.

save_model

ampligraph.utils.**save_model**(*model*, *model_name_path=None*)

Save a trained model to disk.

Examples

```
>>> import numpy as np
>>> from ampligraph.latent_features import ComplEx
>>> from ampligraph.utils import save_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)
>>> y_pred_before = model.predict(np.array([['f', 'y', 'e'], ['b', 'y', 'd']]))
>>> example_name = 'helloworld.pkl'
>>> save_model(model, model_name_path = example_name)
>>> print(y_pred_before)
[-0.29721245, 0.07865551]
```

Parameters

- **model** (EmbeddingModel) – A trained neural knowledge graph embedding model, the model must be an instance of TransE, DistMult, ComplEx, or HolE.
- **model_name_path** (*string*) – The name of the model to be saved. If not specified, a default name model with current datetime is named and saved to the working directory

restore_model

ampligraph.utils.**restore_model**(*model_name_path=None*)

Restore a saved model from disk.

See also *save_model()*.

Examples

```
>>> from ampligraph.utils import restore_model
>>> import numpy as np
>>> example_name = 'helloworld.pkl'
>>> restored_model = restore_model(model_name_path = example_name)
>>> y_pred_after = restored_model.predict(np.array([['f', 'y', 'e'], ['b', 'y', 'd
↪']]))
>>> print(y_pred_after)
[-0.29721245, 0.07865551]
```

Parameters **model_name_path** (*string*) – The name of saved model to be restored. If not specified, the library will try to find the default model in the working directory.

Returns **model** – the neural knowledge graph embedding model restored from disk.

Return type *EmbeddingModel*

Visualization

Functions to visualize embeddings.

create_tensorboard_visualizations(model, loc)	Export embeddings to Tensorboard.

create_tensorboard_visualizations

ampligraph.utils.**create_tensorboard_visualizations**(*model*, *loc*, *labels=None*, *write_metadata=True*, *export_tsv_embeddings=True*)

Export embeddings to Tensorboard.

This function exports embeddings to disk in a format used by TensorBoard and TensorBoard Embedding Projector. The function exports:

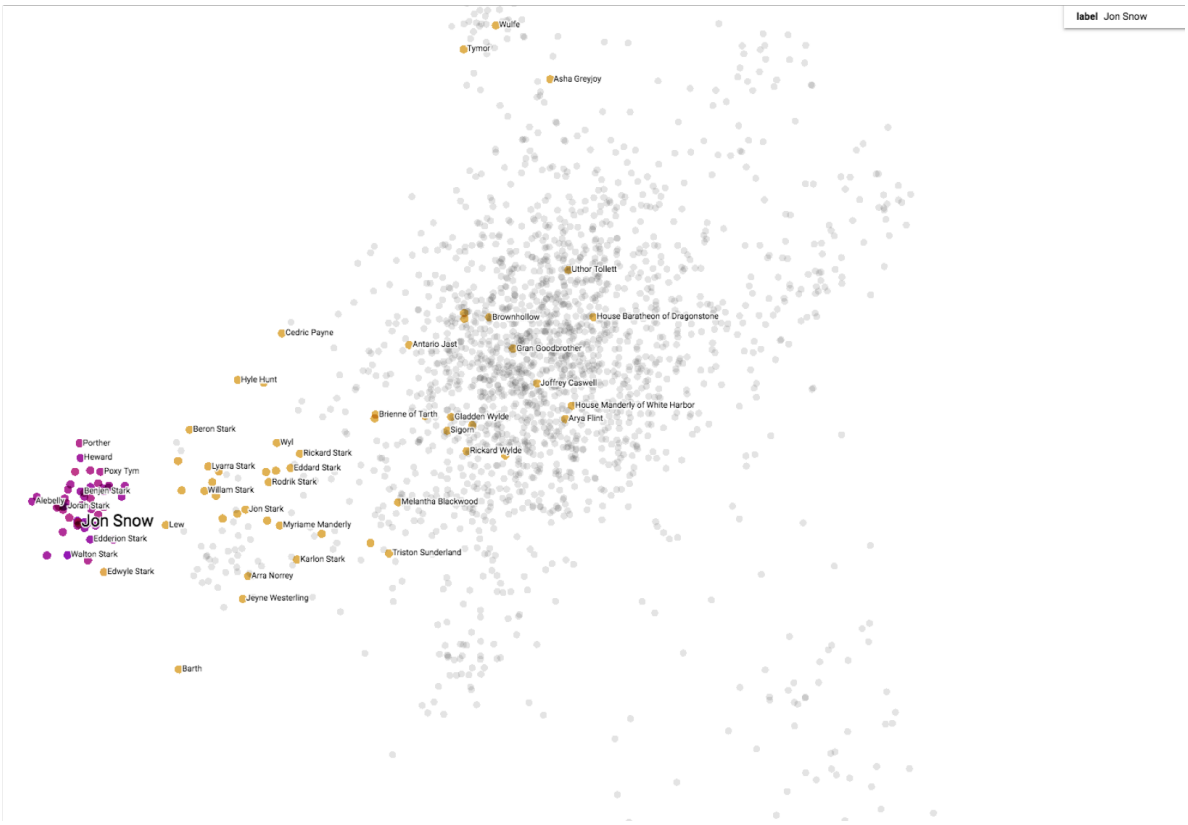
- A number of checkpoint and graph embedding files in the provided location that will allow you to visualize embeddings using Tensorboard. This is generally for use with a local Tensorboard instance.
- a tab-separated file of embeddings `embeddings_projector.tsv`. This is generally used to visualize embeddings by uploading to TensorBoard Embedding Projector.
- embeddings metadata (i.e. the embeddings labels from the original knowledge graph), saved to `metadata.tsv`. Such file can be used in TensorBoard or uploaded to TensorBoard Embedding Projector.

The content of `loc` will look like:

```
tensorboard_files/
    ├── checkpoint
    ├── embeddings_projector.tsv
    ├── graph_embedding.ckpt.data-00000-of-00001
    ├── graph_embedding.ckpt.index
    ├── graph_embedding.ckpt.meta
    ├── metadata.tsv
    └── projector_config.pbtxt
```

Note: A TensorBoard guide is available at this address.

Note: Uploading `embeddings_projector.tsv` and `metadata.tsv` to TensorBoard Embedding Projector will give a result similar to the picture below:



Examples

```
>>> import numpy as np
>>> from ampligraph.latent_features import TransE
>>> from ampligraph.utils import create_tensorboard_visualizations
>>>
>>> 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 = TransE(batches_count=1, seed=555, epochs=20, k=10, loss='pairwise',
>>>                loss_params={'margin':5})
>>> model.fit(X)
>>>
>>> create_tensorboard_visualizations(model, 'tensorboard_files')
```

Parameters

- **model** (EmbeddingModel) – A trained neural knowledge graph embedding model, the model must be an instance of TransE, DistMult, ComplEx, or HolE.
- **loc** (*string*) – Directory where the files are written.

- **labels** (*pd.DataFrame*) – Label(s) for each embedding point in the Tensorboard visualization. Default behaviour is to use the embeddings labels included in the model.
- **export_tsv_embeddings** (*bool (Default: True)*) – If True, will generate a tab-separated file of embeddings at the given path. This is generally used to visualize embeddings by uploading to TensorBoard Embedding Projector.
- **write_metadata** (*bool (Default: True)*) – If True will write a file named 'metadata.tsv' in the same directory as path.

Others

Function to convert a pandas DataFrame with headers into triples.

dataframe_to_triples(X, schema)	Convert DataFrame into triple format.

dataframe_to_triples

ampligraph.utils.**dataframe_to_triples**(*X*, *schema*)

Convert DataFrame into triple format.

Parameters

- **X** (*pandas DataFrame with headers*) –
- **schema** (*List of (subject, relation_name, object) tuples*) – where subject and object are in the headers of the data frame

Examples

```
>>> import pandas as pd
>>> import numpy as np
>>> from ampligraph.utils.model_utils import dataframe_to_triples
>>>
>>> X = pd.read_csv('https://raw.githubusercontent.com/mwaskom/seaborn-data/
↪master/iris.csv')
>>>
>>> schema = [['species', 'has_sepal_length', 'sepal_length']]
>>>
>>> dataframe_to_triples(X, schema)[0]
array(['setosa', 'has_sepal_length', '5.1'], dtype='<U16')
```

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.

Join the conversation on Slack 

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 Developer Notes

Additional documentation on data adapters, AmpliGraph support for large graphs, and further technical details is available here.

3.4.5 Clone and Install in editable mode

Clone the repository and checkout the `develop` branch. Install from source with pip. use the `-e` flag to enable editable mode:

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

3.4.6 Unit Tests

To run all the unit tests:

```
$ pytest tests
```

See pytest documentation for additional arguments.

3.4.7 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`.

3.4.8 Packaging

To build an AmpliGraph custom wheel, do the following:

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

3.5 Examples

These examples show how to get started with AmpliGraph APIs, and cover a range of useful tasks. Note that additional tutorials are also *available*.

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)
```

(continues on next page)

(continued from previous page)

```
                    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
                          '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
                          'corruption_entities':'all', # corrupt using all entities
                          'corrupt_side':'s+o'         # corrupt subject and object␣
↪(but not at once)
                      }
              )



    # 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,
                                 use_default_protocol=True, # corrupt subj and obj␣
↪separately while evaluating
                                 verbose=True)

    # 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.
    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": {
                            # generate corruption using all entities during training
                            "negative_corruption_entities":"all"
                        },
                        "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['train'],
                                                     X_dict['valid'],
                                                     X_dict['test'],
                                                     # Parameter grid
                                                     param_grid,
                                                     # Use filtered set for eval
```

(continues on next page)

(continued from previous page)

```
                                                    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'], embedding_type='entity')
```

3.5.4 Save and restore a model

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

#  Use the trained model to predict
y_pred_before = model.predict(np.array([['f', 'y', 'e'], ['b', 'y', 'd']]))
print(y_pred_before)
#[-0.29721245, 0.07865551]
```

(continues on next page)

(continued from previous page)

```
# Save the model
example_name = "helloworld.pkl"
save_model(model, model_name_path = example_name)

# Restore the model
restored_model = restore_model(model_name_path = example_name)

# Use the restored model to predict
y_pred_after = restored_model.predict(np.array([['f', 'y', 'e'], ['b', 'y', 'd']]))
print(y_pred_after)
# [-0.29721245, 0.07865551]
```

3.5.5 Split dataset into train/test or train/valid/test

```
import numpy as np
from ampligraph.evaluation import train_test_split_no_unseen
from ampligraph.datasets import load_from_csv

'''
Assume we have a knowledge graph stored in my_folder/my_graph.csv,
and that the content of such file is:

a,y,b
f,y,e
b,y,a
a,y,c
c,y,a
a,y,d
c,y,d
b,y,c
f,y,e
'''

# Load the graph in memory
X = load_from_csv('my_folder', 'my_graph.csv', sep=',')

# To split the graph in train and test sets:
# (In this toy example the test set will include only two triples)
X_train, X_test = train_test_split_no_unseen(X, test_size=2)

print(X_train)

'''
X_train:[['a' 'y' 'b']
         ['f' 'y' 'e']
         ['b' 'y' 'a']
         ['c' 'y' 'a']
         ['c' 'y' 'd']
         ['b' 'y' 'c']
         ['f' 'y' 'e']]
'''

print(X_test)
```

(continues on next page)

(continued from previous page)

```
'''
X_test: [['a' 'y' 'c']
         ['a' 'y' 'd']]
'''


# To split the graph in train, validation, and test the method must be called twice:
X_train_valid, X_test = train_test_split_no_unseen(X, test_size=2)
X_train, X_valid = train_test_split_no_unseen(X_train_valid, test_size=2)

print(X_train)
'''
X_train:  [['a' 'y' 'b']
           ['b' 'y' 'a']
           ['c' 'y' 'd']
           ['b' 'y' 'c']
           ['f' 'y' 'e']]
'''

print(X_valid)
'''
X_valid:  [['f' 'y' 'e']
           ['c' 'y' 'a']]
'''

print(X_test)
'''
X_test:  [['a' 'y' 'c']
          ['a' 'y' 'd']]
'''
```

3.5.6 Clustering and projectings embeddings into 2D space

Embedding training

```
import numpy as np
import pandas as pd
import requests

from ampligraph.datasets import load_from_csv
from ampligraph.latent_features import ComplEx
from ampligraph.evaluation import evaluate_performance
from ampligraph.evaluation import mr_score, mrr_score, hits_at_n_score
from ampligraph.evaluation import train_test_split_no_unseen

# International football matches triples
url = 'https://ampligraph.s3-eu-west-1.amazonaws.com/datasets/football.csv'
open('football.csv', 'wb').write(requests.get(url).content)
X = load_from_csv('.', 'football.csv', sep=',')[:, 1:]

# Train test split
X_train, X_test = train_test_split_no_unseen(X, test_size=10000)

# ComplEx model
```

(continues on next page)

(continued from previous page)

```
model = ComplEx(batches_count=50,
                epochs=300,
                k=100,
                eta=20,
                optimizer='adam',
                optimizer_params={'lr':1e-4},
                loss='multiclass_nll',
                regularizer='LP',
                regularizer_params={'p':3, 'lambda':1e-5},
                seed=0,
                verbose=True)

model.fit(X_train)
```

Embedding evaluation

```
filter_triples = np.concatenate((X_train, X_test))
ranks = evaluate_performance(X_test,
                             model=model,
                             filter_triples=filter_triples,
                             use_default_protocol=True,
                             verbose=True)

mr = mr_score(ranks)
mrr = mrr_score(ranks)

print("MRR: %.2f" % (mrr))
print("MR: %.2f" % (mr))

hits_10 = hits_at_n_score(ranks, n=10)
print("Hits@10: %.2f" % (hits_10))
hits_3 = hits_at_n_score(ranks, n=3)
print("Hits@3: %.2f" % (hits_3))
hits_1 = hits_at_n_score(ranks, n=1)
print("Hits@1: %.2f" % (hits_1))
'''
MRR: 0.25
MR: 4927.33
Hits@10: 0.35
Hits@3: 0.28
Hits@1: 0.19
'''
```

Clustering and 2D projections

Please install lib adjustText first with `pip install adjustText`. For incf.countryutils, do the following steps:

```
git clone https://github.com/wyldebeast-wunderliebe/incf.countryutils.git
cd incf.countryutils
pip install .
```

incf.countryutils is used to map countries to the corresponding continents.

```
import re
from sklearn.decomposition import PCA
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
import seaborn as sns
from adjustText import adjust_text
from incf.countryutils import transformations
from ampligraph.discovery import find_clusters

# Get the teams entities and their corresponding embeddings
triples_df = pd.DataFrame(X, columns=['s', 'p', 'o'])
teams = triples_df.s[triples_df.s.str.startswith('Team')].unique()
team_embeddings = dict(zip(teams, model.get_embeddings(teams)))
team_embeddings_array = np.array([i for i in team_embeddings.values()])

# Project embeddings into 2D space via PCA
embeddings_2d = PCA(n_components=2).fit_transform(team_embeddings_array)

# Cluster embeddings (on the original space)
clustering_algorithm = KMeans(n_clusters=6, n_init=100, max_iter=500, random_state=0)
clusters = find_clusters(teams, model, clustering_algorithm, mode='entity')

# This function maps country to continent
def cn_to_ctn(country):
    try:
        original_name = ' '.join(re.findall('[A-Z][^A-Z]*', country[4:]))
        return transformations.cn_to_ctn(original_name)
    except KeyError:
        return "unk"

plot_df = pd.DataFrame({"teams": teams,
                        "embedding1": embeddings_2d[:, 0],
                        "embedding2": embeddings_2d[:, 1],
                        "continent": pd.Series(teams).apply(cn_to_ctn),
                        "cluster": "cluster" + pd.Series(clusters).astype(str)})

# Top 20 teams in 2019 according to FIFA rankings
top20teams = ["TeamBelgium", "TeamFrance", "TeamBrazil", "TeamEngland", "TeamPortugal
↪",
              "TeamCroatia", "TeamSpain", "TeamUruguay", "TeamSwitzerland",
↪"TeamDenmark",
              "TeamArgentina", "TeamGermany", "TeamColombia", "TeamItaly",
↪"TeamNetherlands",
              "TeamChile", "TeamSweden", "TeamMexico", "TeamPoland", "TeamIran"]

np.random.seed(0)

# Plot 2D embeddings with country labels
def plot_clusters(hue):
    plt.figure(figsize=(12, 12))
    plt.title("{} embeddings".format(hue).capitalize())
    ax = sns.scatterplot(data=plot_df[plot_df.continent!="unk"],
                         x="embedding1", y="embedding2", hue=hue)
    texts = []
    for i, point in plot_df.iterrows():
        if point["teams"] in top20teams or np.random.random() < 0.1:
            texts.append(plt.text(point['embedding1']+0.02,
```

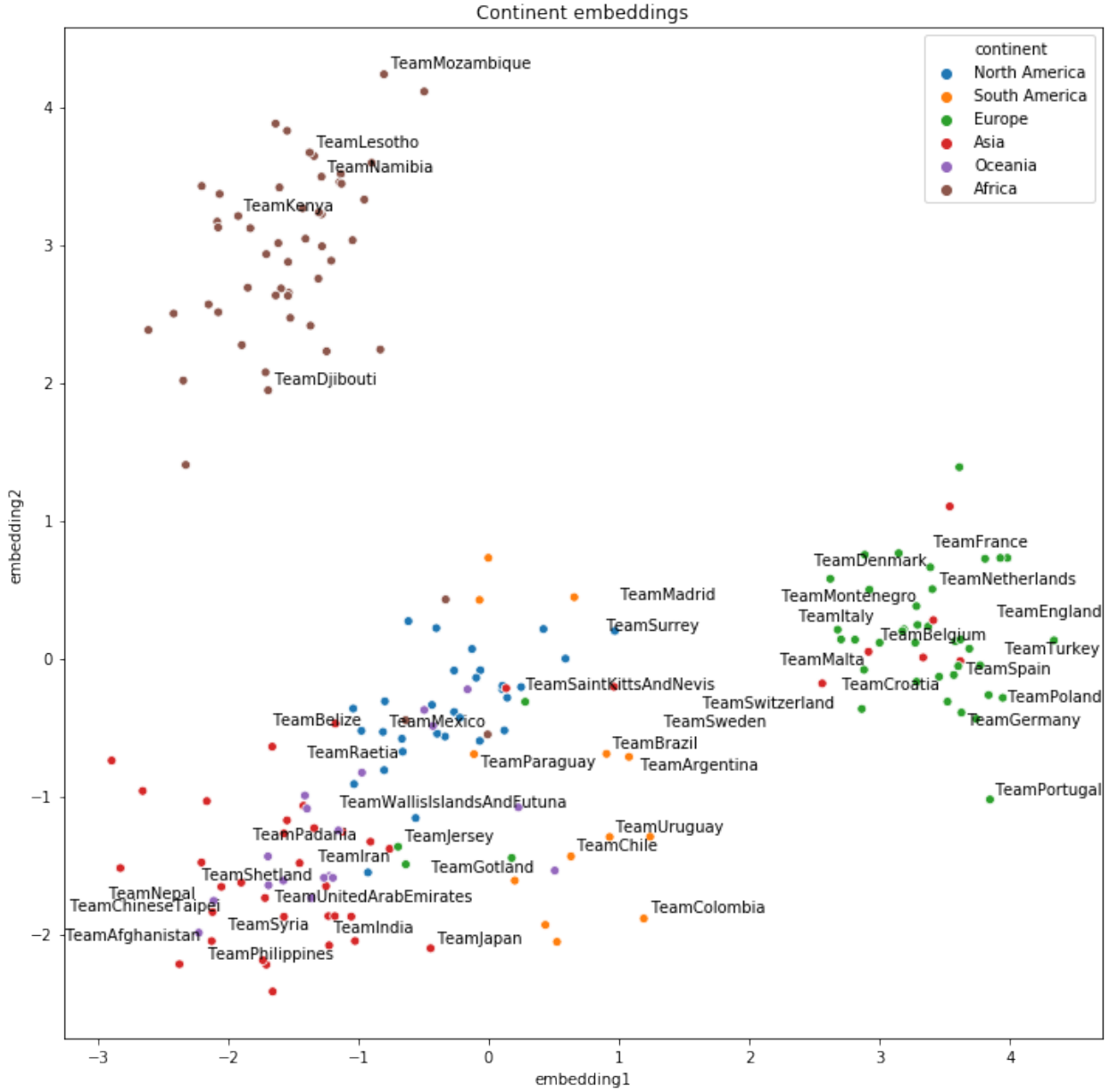
(continues on next page)

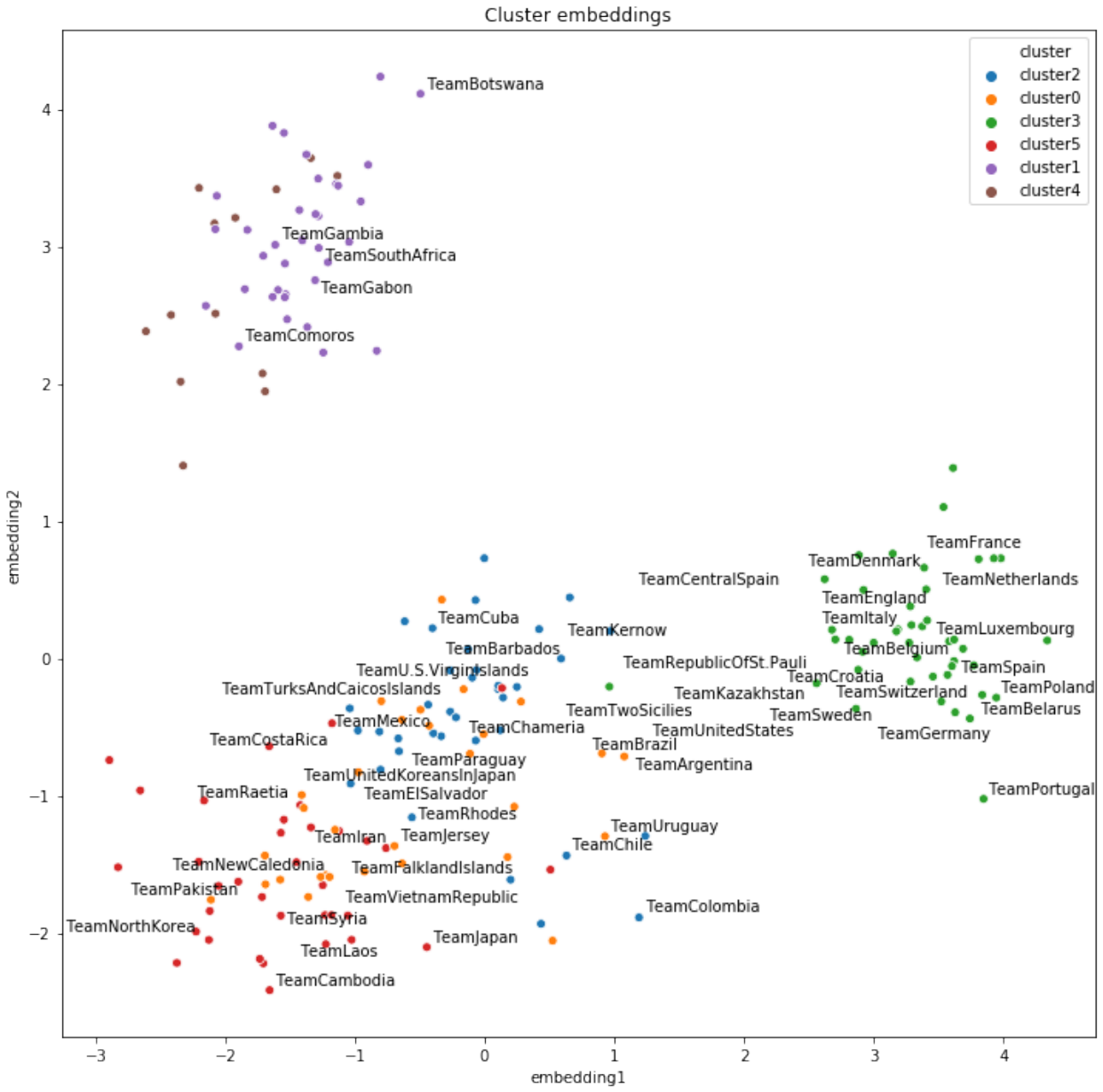
(continued from previous page)

```
                        point['embedding2']+0.01,
                        str(point["teams"])))
    adjust_text(texts)
```

Results visualization

```
plot_clusters("continent")
plot_clusters("cluster")
```





3.6 Tutorials

The following Jupyter notebooks will guide you through the most important features of AmpliGraph:

- AmpliGraph basics: training, saving and restoring a model, evaluating a model, discover new links, visualize embeddings. [Jupyter notebook] [Colab notebook]
- Link-based clustering and classification: how to use the knowledge embeddings generated by a graph of international football matches in clustering and classification tasks. [Jupyter notebook] [Colab notebook]

Additional examples and code snippets are *available here*.

3.7 Performance

3.7.1 Predictive Performance

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

Note: On ConvE Evaluation. Results reported in the literature for ConvE are based on the alternative *1-N* evaluation protocol which requires that reciprocal relations are added to the dataset [DMSR18]:

$$D \leftarrow (D, D_{recip})$$

$$D_{recip} \leftarrow \{ (o, p_r, s) \,|\, \forall x \in D, x = (s, p, o) \}$$

During training each unique pair of subject and predicate can predict all possible object scores for that pairs, and therefore object corruptions evaluation can be performed with a single forward pass:

$$ConvE(s, p, o)$$

In the standard corruption procedure the subject entity is replaced by corruptions:

$$ConvE(s_{corr}, p, o),$$

However in the *1-N* protocol subject corruptions are interpreted as object corruptions of the reciprocal relation:

$$ConvE(o, p_r, s_{corr})$$

To reproduce the results reported in the literature using the *1-N* evaluation protocol, add reciprocal relations by specifying `add_reciprocal_rels` in the dataset loader function, e.g. `load_fb15k(add_reciprocal_rels=True)`, and run the evaluation protocol with object corruptions by specifying `corrupt_sides='o'`.

Results obtained with the standard evaluation protocol are labeled *ConvE*, while those obtained with the *1-N* protocol are marked *ConvE(1-N)*.

3.7.2 FB15K-237

Model	MR	MRR	Hits@1	Hits@3	Hits@10	Hyperparameters
TransE	208	0.31	0.22	0.35	0.50	k: 400; epochs: 4000; eta: 30; loss: multiclass_nll; regularizer: LP; regularizer_params: lambda: 0.0001; p: 2; optimizer: adam; optimizer_params: lr: 0.0001; embedding_model_params: norm: 1; normalize_ent_emb: false; seed: 0; batches_count: 64;
DistMult	199	0.31	0.22	0.35	0.49	k: 300; epochs: 4000; eta: 50; loss: multiclass_nll; regularizer: LP; regularizer_params: lambda: 0.0001; p: 3; optimizer: adam; optimizer_params: lr: 0.00005; seed: 0; batches_count: 50; normalize_ent_emb: false;
ComplEx	184	0.32	0.23	0.35	0.50	k: 350; epochs: 4000; eta: 30; loss: multiclass_nll; optimizer: adam; optimizer_params: lr: 0.00005; seed: 0; regularizer: LP; regularizer_params: lambda: 0.0001; p: 3; batches_count: 64;
HolE	184	0.31	0.22	0.34	0.49	k: 350; epochs: 4000; eta: 50; loss: multiclass_nll; regularizer: LP; regularizer_params: lambda: 0.0001; p: 2; optimizer: adam; optimizer_params: lr: 0.0001; seed: 0; batches_count: 64;
ConvKB	327	0.23	0.15	0.25	0.40	k: 200; epochs: 500; eta: 10; loss: multiclass_nll; loss_params: {} optimizer: adam; optimizer_params: lr: 0.0001; embedding_model_params:{ num_filters: 32, filter_sizes: 1, dropout: 0.1}; seed: 0; batches_count: 300;
ConvE	1060	0.26	0.19	0.28	0.38	k: 200; epochs: 4000; loss: bce; loss_params: {label_smoothing=0.1} optimizer: adam; optimizer_params: lr: 0.0001; embedding_model_params:{ conv_filters: 32, conv_kernel_size: 3, dropout_embed: 0.2, dropout_conv: 0.1, dropout_dense: 0.3, use_batchnorm: True, use_bias: True}; seed: 0; batches_count: 100;
ConvE (CWN)	234	0.32	0.23	0.35	0.50	k: 200; epochs: 4000; loss: bce; loss_params: {label_smoothing=0.1} optimizer: adam; optimizer_params: lr: 0.0001; embedding_model_params:{ conv_filters: 32, conv_kernel_size: 3, dropout_embed: 0.2, dropout_conv: 0.1, dropout_dense: 0.3, use_batchnorm: True, use_bias: True}; seed: 0; batches_count: 100;

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).

3.7.3 WN18RR

Model	MR	MRR	Hits@1	Hits@3	Hits@10	Hyperparameters
TransE	2692	0.22	0.03	0.37	0.54	k: 350; epochs: 4000; eta: 30; loss: multiclass_nll; optimizer: adam; optimizer_params: lr: 0.0001; regularizer: LP; regularizer_params: lambda: 0.0001; p: 2; seed: 0; normalize_ent_emb: false; embedding_model_params: norm: 1; batches_count: 150;
DistMult	5531	0.47	0.43	0.48	0.53	k: 350; epochs: 4000; eta: 30; loss: multiclass_nll; optimizer: adam; optimizer_params: lr: 0.0001; regularizer: LP; regularizer_params: lambda: 0.0001; p: 2; seed: 0; normalize_ent_emb: false; batches_count: 100;
ComplEx	4177	0.51	0.46	0.53	0.58	k: 200; epochs: 4000; eta: 20; loss: multiclass_nll; loss_params: margin: 1; optimizer: adam; optimizer_params: lr: 0.0005; seed: 0; regularizer: LP; regularizer_params: lambda: 0.05; p: 3; batches_count: 10;
HolE	7028	0.47	0.44	0.48	0.53	k: 200; epochs: 4000; eta: 20; loss: self_adversarial; loss_params: margin: 1; optimizer: adam; optimizer_params: lr: 0.0005; seed: 0; batches_count: 50;
ConvKB	3652	0.39	0.33	0.42	0.48	k: 200; epochs: 500; eta: 10; loss: multiclass_nll; loss_params: {} optimizer: adam; optimizer_params: lr: 0.0001; embedding_model_params:{ num_filters: 32, filter_sizes: 1, dropout: 0.1}; seed: 0; batches_count: 300;
ConvE	5346	0.45	0.42	0.47	0.52	k: 200; epochs: 4000; loss: bce; loss_params: {label_smoothing=0.1} optimizer: adam; optimizer_params: lr: 0.0001; embedding_model_params:{ conv_filters: 32, conv_kernel_size: 3, dropout_embed: 0.2, dropout_conv: 0.1, dropout_dense: 0.3, use_batchnorm: True, use_bias: True}; seed: 0; batches_count: 100;
ConvE (1-N)	4842	0.48	0.45	0.49	0.54	k: 200; epochs: 4000; loss: bce; loss_params: {label_smoothing=0.1} optimizer: adam; optimizer_params: lr: 0.0001; embedding_model_params:{ conv_filters: 32, conv_kernel_size: 3, dropout_embed: 0.2, dropout_conv: 0.1, dropout_dense: 0.3, use_batchnorm: True, use_bias: True}; seed: 0; batches_count: 100;

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).

3.7.4 YAGO3-10

Model	MR	MRR	Hits@1	Hits@3	Hits@10	Hyperparameters
TransE	1264	0.51	0.41	0.57	0.67	k: 350; epochs: 4000; eta: 30; loss: multiclass_nll; optimizer: adam; optimizer_params: lr: 0.0001; regularizer: LP; regularizer_params: lambda: 0.0001; p: 2; embedding_model_params: norm: 1; normalize_ent_emb: false; seed: 0; batches_count: 100;
DistMult	1107	0.50	0.41	0.55	0.66	k: 350; epochs: 4000; eta: 50; loss: multiclass_nll; optimizer: adam; optimizer_params: lr: 5e-05; regularizer: LP; regularizer_params: lambda: 0.0001; p: 3; seed: 0; normalize_ent_emb: false; batches_count: 100;
ComplEx	1227	0.49	0.40	0.54	0.66	k: 350; epochs: 4000; eta: 30; loss: multiclass_nll; optimizer: adam; optimizer_params: lr: 5e-05; regularizer: LP; regularizer_params: lambda: 0.0001; p: 3; seed: 0; batches_count: 100
HolE	6776	0.50	0.42	0.56	0.65	k: 350; epochs: 4000; eta: 30; loss: self_adversarial; loss_params: alpha: 1; margin: 0.5; optimizer: adam; optimizer_params: lr: 0.0001; seed: 0; batches_count: 100
ConvKB	2820	0.30	0.21	0.34	0.50	k: 200; epochs: 500; eta: 10; loss: multiclass_nll; loss_params: {} optimizer: adam; optimizer_params: lr: 0.0001; embedding_model_params:{ num_filters: 32, filter_sizes: 1, dropout: 0.1}; seed: 0; batches_count: 3000;
ConvE	6063	0.40	0.33	0.42	0.53	k: 300; epochs: 4000; loss: bce; loss_params: {label_smoothing=0.1} optimizer: adam; optimizer_params: lr: 0.0001; embedding_model_params:{ conv_filters: 32, conv_kernel_size: 3, dropout_embed: 0.2, dropout_conv: 0.1, dropout_dense: 0.3, use_batchnorm: True, use_bias: True}; seed: 0; batches_count: 300;
ConvE (1-N)	2741	0.55	0.48	0.60	0.69	k: 300; epochs: 4000; loss: bce; loss_params: {label_smoothing=0.1} optimizer: adam; optimizer_params: lr: 0.0001; embedding_model_params:{ conv_filters: 32, conv_kernel_size: 3, dropout_embed: 0.2, dropout_conv: 0.1, dropout_dense: 0.3, use_batchnorm: True, use_bias: True}; seed: 0; batches_count: 300;

Note: YAGO3-10 validation and test sets include triples with entities that do not occur in the training set. We found 22 unseen entities in the validation set and 18 in the test set. In the experiments we excluded the triples where such entities appear (22 triples in from the validation set and 18 from the test set).

3.7.5 FB15K

Warning: The dataset includes a large number of inverse relations, and its use in experiments has been deprecated. Use FB15k-237 instead.

Model	MR	MRR	Hits@1	Hits@3	Hits@10	Hyperparameters
TransE	44	0.63	0.50	0.73	0.85	k: 150; epochs: 4000; eta: 10; loss: multiclass_nll; optimizer: adam; optimizer_params: lr: 5e-5; regularizer: LP; regularizer_params: lambda: 0.0001; p: 3; embedding_model_params: norm: 1; normalize_ent_emb: false; seed: 0; batches_count: 100;
DistMult	179	0.78	0.74	0.82	0.86	k: 200; epochs: 4000; eta: 20; loss: self_adversarial; loss_params: margin: 1; optimizer: adam; optimizer_params: lr: 0.0005; seed: 0; normalize_ent_emb: false; batches_count: 50;
ComplEx	184	0.80	0.76	0.82	0.86	k: 200; epochs: 4000; eta: 20; loss: self_adversarial; loss_params: margin: 1; optimizer: adam; optimizer_params: lr: 0.0005; seed: 0; batches_count: 100;
HolE	216	0.80	0.76	0.83	0.87	k: 200; epochs: 4000; eta: 20; loss: self_adversarial; loss_params: margin: 1; optimizer: adam; optimizer_params: lr: 0.0005; seed: 0; batches_count: 50;
ConvKB	331	0.65	0.55	0.71	0.82	k: 200; epochs: 500; eta: 10; loss: multiclass_nll; loss_params: {} optimizer: adam; optimizer_params: lr: 0.0001; embedding_model_params:{ num_filters: 32, filter_sizes: 1, dropout: 0.1}; seed: 0; batches_count: 300;
ConvE	385	0.50	0.42	0.52	0.66	k: 300; epochs: 4000; loss: bce; loss_params: {label_smoothing=0.1} optimizer: adam; optimizer_params: lr: 0.0001; embedding_model_params:{ conv_filters: 32, conv_kernel_size: 3, dropout_embed: 0.2, dropout_conv: 0.1, dropout_dense: 0.3, use_batchnorm: True, use_bias: True}; seed: 0; batches_count: 100;
ConvE (1-N)	55	0.80	0.74	0.84	0.89	k: 300; epochs: 4000; loss: bce; loss_params: {label_smoothing=0.1} optimizer: adam; optimizer_params: lr: 0.0001; embedding_model_params:{ conv_filters: 32, conv_kernel_size: 3, dropout_embed: 0.2, dropout_conv: 0.1, dropout_dense: 0.3, use_batchnorm: True, use_bias: True}; seed: 0; batches_count: 100;

3.7.6 WN18

Warning: The dataset includes a large number of inverse relations, and its use in experiments has been deprecated. Use WN18RR instead.

Model	MR	MRR	Hits@1	Hits@3	Hits@10	Hyperparameters
TransE	260	0.66	0.44	0.88	0.95	k: 150; epochs: 4000; eta: 10; loss: multiclass_nll; optimizer: adam; optimizer_params: lr: 5e-5; regularizer: LP; regularizer_params: lambda: 0.0001; p: 3; embedding_model_params: norm: 1; normalize_ent_emb: false; seed: 0; batches_count: 100;
DistMult	675	0.82	0.73	0.92	0.95	k: 200; epochs: 4000; eta: 20; loss: nll; loss_params: margin: 1; optimizer: adam; optimizer_params: lr: 0.0005; seed: 0; normalize_ent_emb: false; batches_count: 50;
ComplEx	726	0.94	0.94	0.95	0.95	k: 200; epochs: 4000; eta: 20; loss: nll; loss_params: margin: 1; optimizer: adam; optimizer_params: lr: 0.0005; seed: 0; batches_count: 50;
HolE	665	0.94	0.93	0.94	0.95	k: 200; epochs: 4000; eta: 20; loss: self_adversarial; loss_params: margin: 1; optimizer: adam; optimizer_params: lr: 0.0005; seed: 0; batches_count: 50;
ConvKB	331	0.80	0.69	0.90	0.94	k: 200; epochs: 500; eta: 10; loss: multiclass_nll; loss_params: {} optimizer: adam; optimizer_params: lr: 0.0001; embedding_model_params:{ num_filters: 32, filter_sizes: 1, dropout: 0.1}; seed: 0; batches_count: 300;
ConvE	492	0.93	0.91	0.94	0.95	k: 300; epochs: 4000; loss: bce; loss_params: {label_smoothing=0.1} optimizer: adam; optimizer_params: lr: 0.0001; embedding_model_params:{ conv_filters: 32, conv_kernel_size: 3, dropout_embed: 0.2, dropout_conv: 0.1, dropout_dense: 0.3, use_batchnorm: True, use_bias: True}; seed: 0; batches_count: 100;
ConvE (1-N)	436	0.95	0.93	0.95	0.95	k: 300; epochs: 4000; loss: bce; loss_params: {label_smoothing=0.1} optimizer: adam; optimizer_params: lr: 0.0001; embedding_model_params:{ conv_filters: 32, conv_kernel_size: 3, dropout_embed: 0.2, dropout_conv: 0.1, dropout_dense: 0.3, use_batchnorm: True, use_bias: True}; seed: 0; batches_count: 100;

To reproduce the above results:

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

Note: Running `predictive_performance.py` on all datasets, for all models takes ~115 hours on an Intel Xeon Gold 6142, 64 GB Ubuntu 16.04 box equipped with a Tesla V100 16GB. The long running time is mostly due to the early stopping configuration (see section below).

Note: All of the experiments above were conducted with early stopping on half the validation set. Typically, the validation set can be found in `X['valid']`. We only used half the validation set so the other half is available for hyperparameter tuning.

The exact early stopping configuration is as follows:

- x_valid: validation[::2]
- criteria: mrr
- x_filter: train + validation + test
- stop_interval: 4

- burn_in: 0
- check_interval: 50

Note that early stopping adds a significant computational burden to the learning procedure. To lessen it, you may either decrease the validation set, the stop interval, the check interval, or increase the burn in.

Note: Due to a combination of model and dataset size it is not possible to evaluate Yago3-10 with ConvKB on the GPU. The fastest way to replicate the results above is to train ConvKB with Yago3-10 on a GPU using the hyper-parameters described above (~15hrs on GTX 1080Ti), and then evaluate the model in CPU only mode (~15 hours on Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz).

Note: ConvKB with early-stopping evaluation protocol does not fit into GPU memory, so instead is just trained for a set number of epochs.

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,yago310}]
                                 [-m {complex,transe,distmult,hole,convkb,conve}]

optional arguments:
  -h, --help            show this help message and exit
  -d {fb15k,fb15k-237,wn18,wn18rr,yago310}, --dataset {fb15k,fb15k-237,wn18,wn18rr,
↪yago310}
  -m {complex,transe,distmult,hole,convkb,conve}, --model {complex,transe,distmult,
↪hole,convkb,conve}
```

3.7.7 Runtime Performance

Training the models on FB15K-237 (`k=100, eta=10, batches_count=100, loss=multiclass_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	1.33
TransE	1.22
DistMult	1.20
HolE	1.30
ConvKB	2.83
ConvE	1.13

Note: ConvE is trained with `bce` loss instead of `multiclass_nll`.

3.8 Bibliography

3.9 Changelog

3.9.1 1.3.1

18 Mar 2020

- Minor bug fix in ConvE (#189)

3.9.2 1.3.0

9 Mar 2020

- ConvE model Implementation (#178)
- Changes to evaluate_performance API (#183)
- Option to add reciprocal relations (#181)
- Minor fixes to ConvKB (#168, #167)
- Minor fixes in large graph mode (#174, #172, #169)
- Option to skip unseen entities checks when train_test_split is used (#163)
- Stability of NLL losses (#170)
- ICLR-20 calibration paper experiments added in branch paper/ICLR-20

3.9.3 1.2.0

22 Oct 2019

- Probability calibration using Platt scaling, both with provided negatives or synthetic negative statements (#131)
- Added ConvKB model
- Added WN11, FB13 loaders (datasets with ground truth positive and negative triples) (#138)
- Continuous integration with CircleCI, integrated on GitHub (#127)
- Refactoring of models into separate files (#104)
- Fixed bug where the number of epochs did not exactly match the provided number by the user (#135)
- Fixed some bugs on RandomBaseline model (#133, #134)
- Fixed some bugs on discover_facts with strategies “exhaustive” and “graph_degree”
- Fixed bug on subsequent calls of model.predict on the GPU (#137)

3.9.4 1.1.0

16 Aug 2019

- Support for large number of entities (#61, #113)
- Faster evaluation protocol (#74)
- New Knowledge discovery APIs: discover facts, clustering, near-duplicates detection, topn query (#118)
- API change: model.predict() does not return ranks anymore
- API change: friendlier ranking API output (#101)
- Implemented nuclear-3 norm (#23)
- Jupyter notebook tutorials: AmpliGraph basics (#67) and Link-based clustering
- Random search for hyper-parameter tuning (#106)
- Additional initializers (#112)
- Experiment campaign with multiclass-loss
- System-wide bugfixes and minor improvements

3.9.5 1.0.3

7 Jun 2019

- Fixed regression in RandomBaseline (#94)
- Added TensorBoard Embedding Projector support (#86)
- Minor bugfixing (#87, #47)

3.9.6 1.0.2

19 Apr 2019

- Added multiclass loss (#24 and #22)
- Updated the negative generation to speed up evaluation for default protocol.(#74)
- Support for visualization of embeddings using Tensorboard (#16)
- Save models with custom names. (#71)
- Quick fix for the overflow issue for datasets with a million entities. (#61)
- Fixed issues in train_test_split_no_unseen API and updated api (#68)
- Added unit test cases for better coverage of the code(#75)
- Corrupt_sides : can now generate corruptions for training on both sides, or only on subject or object
- Better error messages
- Reduced logging verbosity
- Added YAGO3-10 experiments
- Added MD5 checksum for datasets (#47)
- Addressed issue of ambiguous dataset loaders (#59)

- Renamed 'type' parameter in models.get_embeddings to fix masking built-in function
- Updated String comparison to use equality instead of identity.
- Moved save_model and restore_model to ampligraph.utils (but existing API will remain for several releases).
- Other minor issues (#63, #64, #65, #66)

3.9.7 1.0.1

22 Mar 2019

- evaluation protocol now ranks object and subjects corruptions separately
- Corruption generation can now use entities from current batch only
- FB15k-237, WN18RR loaders filter out unseen triples by default
- Removed some unused arguments
- Improved documentation
- Minor bugfixing

3.9.8 1.0.0

16 Mar 2019

- TransE
- DistMult
- ComplEx
- FB15k, WN18, FB15k-237, WN18RR, YAGO3-10 loaders
- generic loader for csv files
- RDF, ntriples loaders
- Learning to rank evaluation protocol
- Tensorflow-based negatives generation
- save/restore capabilities for models
- 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

3.10 About

AmpliGraph is maintained by Accenture Labs Dublin.

3.10.1 Contact us

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

Join the conversation on Slack 

3.10.2 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
          Nicholas McCarthy and
          Pedro Tabacof},
 title = {{AmpliGraph: a Library for Representation Learning on Knowledge Graphs}},
 month = mar,
 year  = 2019,
 doi   = {10.5281/zenodo.2595043},
 url   = {https://doi.org/10.5281/zenodo.2595043}
}
```

3.10.3 Contributors

Active contributors (in alphabetical order)

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

Past contributors

- Pedro Tabacof

3.10.4 License

AmpliGraph is licensed under the Apache 2.0 License.

BIBLIOGRAPHY

[aC15] Danqi and Chen. Observed versus latent features for knowledge base and text inference. In *3rd Workshop on Continuous Vector Space Models and Their Compositionality*. ACL - Association for Computational Linguistics, July 2015. URL: https://www.microsoft.com/en-us/research/publication/observed-versus-latent-features-for-knowledge-base-and-text-inference/.

[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.

[BB12] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

[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.

[BEP+08] Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 1247–1250. AcM, 2008.

[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.

[GB10] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, 249–256. 2010.

[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.

[KBK17] Rudolf Kadlec, Ondrej Bajgar, and Jan Kleindienst. Knowledge base completion: baselines strike back. *CoRR*, 2017. URL: http://arxiv.org/abs/1705.10744, arXiv:1705.10744.

[LUO18] Timothee Lacroix, Nicolas Usunier, and Guillaume Obozinski. Canonical tensor decomposition for knowledge base completion. In *International Conference on Machine Learning*, 2869–2878. 2018.

[LJ18] Lisha Li and Kevin Jamieson. Hyperband: a novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research*, 18:1–52, 2018.

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

[Mil95] George A Miller. Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.

[NNNP18] Dai Quoc Nguyen, Tu Dinh Nguyen, Dat Quoc Nguyen, and Dinh Phung. A Novel Embedding Model for Knowledge Base Completion Based on Convolutional Neural Network. In *Proceedings of the 16th Annual Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, 327–333. 2018.

[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.

[P+99] John Platt and others. Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods. *Advances in large margin classifiers*, 10(3):61–74, 1999.

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

[SCMN13] Richard Socher, Danqi Chen, Christopher D Manning, and Andrew Ng. Reasoning with neural tensor networks for knowledge base completion. In *Advances in neural information processing systems*, 926–934. 2013.

[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.

[TC20] Pedro Tabacof and Luca Costabello. Probability Calibration for Knowledge Graph Embedding Models. In *ICLR*. 2020.

[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

B

C

D

E

F

G

H

I

L

M

N

P

Q

R

S

T

X