
AmpliGraph

Release 2.1-dev

Luca Costabello - Accenture Labs Dublin

Feb 28, 2024

CONTENTS:

1	Key Features	3
2	Modules	5
3	How to Cite	7
	Bibliography	75
	Python Module Index	77
	Index	79

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

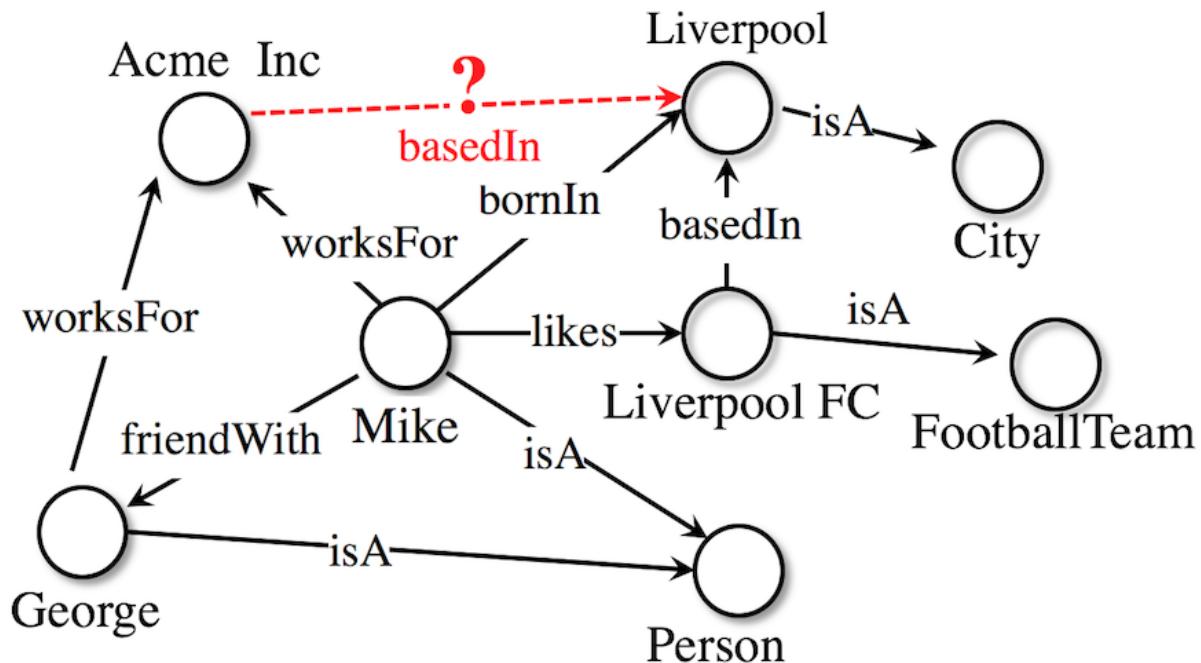
Go to the GitHub repository



Join the conversation on Slack



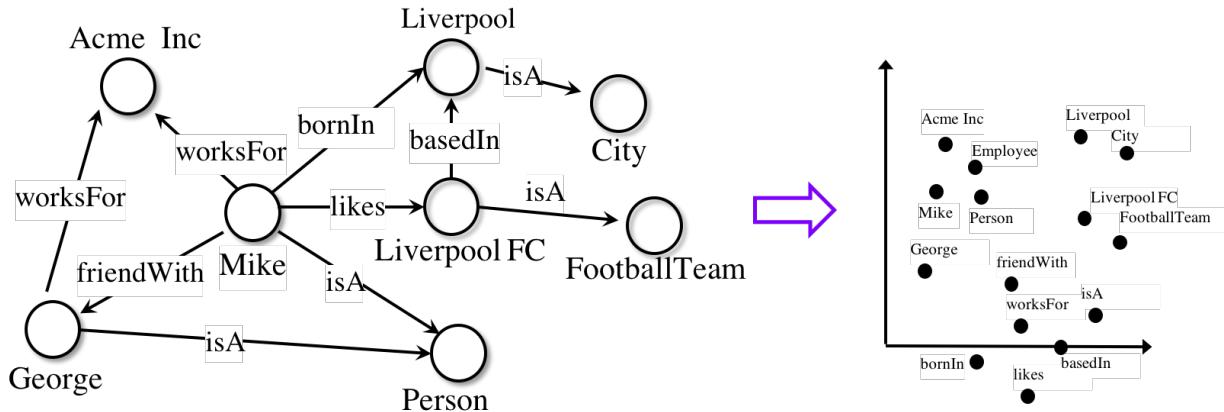
AmpliGraph is a suite of neural machine learning models for relational Learning, a branch of machine learning that deals with supervised learning on knowledge graphs.



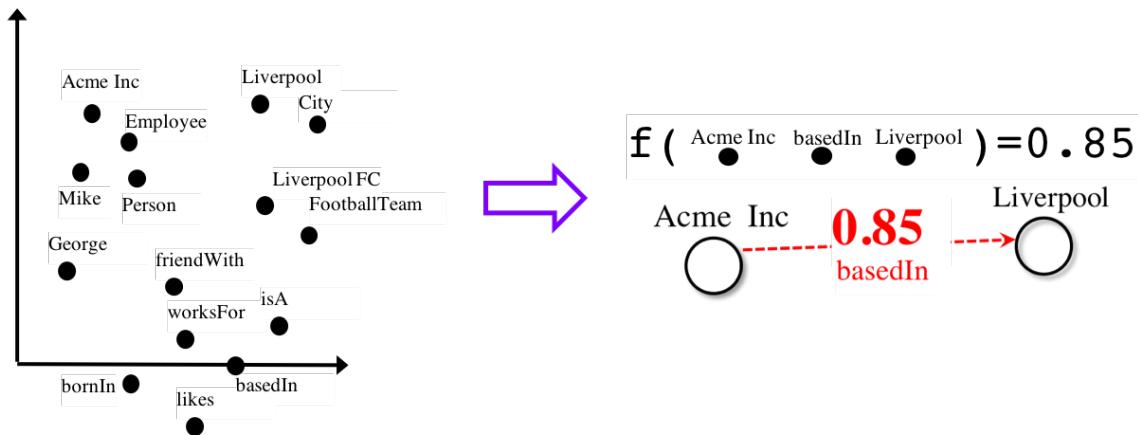
Use AmpliGraph if you need to:

- 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. The new version AmpliGraph 2 APIs are in Keras style, making the user experience even smoother. * **GPU-Ready:** AmpliGraph is built on top of TensorFlow 2, 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 offers **TransE**, **DistMult**, **ComplEx**, **Hole**, **RotateE** (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).
- **Compat**: submodule that extends the compatibility of AmpliGraph APIs to those of AmpliGraph 1.x for the user already familiar with them.

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
              Alberto Bernardi and
              Adrianna Janik and
              Aldan Creo 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 }
}
```

3.1 Installation

3.1.1 Prerequisites

- Linux, macOS, Windows
- Python 3.8

3.1.2 Provision a Virtual Environment

To provision a virtual environment for installing AmpliGraph, any option can work; here we will give provide the instruction for using `venv` and `Conda`.

`venv`

The first step is to create and activate the virtual environment.

```
python3.8 -m venv PATH/TO/NEW/VIRTUAL_ENVIRONMENT  
source PATH/TO/NEW/VIRTUAL_ENVIRONMENT/bin/activate
```

Once this is done, we can proceed with the installation of TensorFlow 2:

```
pip install "tensorflow==2.9.0"
```

If you are installing Tensorflow on MacOS, instead of the following please use:

```
pip install "tensorflow-macos==2.9.0"
```

IMPORTANT: the installation of TensorFlow can be tricky on Mac OS with the Apple silicon chip. Though `venv` can provide a smooth experience, we invite you to refer to the dedicated section down below and consider using `conda` if some issues persist in alignment with the [Tensorflow Plugin page](#) on Apple developer site.

`Conda`

The first step is to create and activate the virtual environment.

```
conda create --name ampligraph python=3.8  
source activate ampligraph
```

Once this is done, we can proceed with the installation of TensorFlow 2, which can be done through `pip` or `conda`.

```
pip install "tensorflow==2.9.0"
```

or

```
conda install "tensorflow==2.9.0"
```

Install TensorFlow 2 for Mac OS M1 chip

When installing TensorFlow 2 for Mac OS with Apple silicon chip we recommend to use a `conda` environment.

```
conda create --name ampligraph python=3.8  
source activate ampligraph
```

After having created and activated the virtual environment, run the following to install Tensorflow.

```
conda install -c apple tensorflow-deps  
pip install --user tensorflow-macos==2.9.0  
pip install --user tensorflow-metal==0.6
```

In case of problems with the installation or for further details, refer to [Tensorflow Plugin page](#) on the official Apple developer website.

3.1.3 Install AmpliGraph

Once the installation of Tensorflow is complete, we can proceed with the installation of AmpliGraph.

To install the latest stable release from pip:

```
pip install ampligraph
```

To sanity check the installation, run the following:

```
>>> import ampligraph
>>> ampligraph.__version__
'2.1.0'
```

If instead you want the most recent development version, you can clone the repository from [GitHub](#), install AmpliGraph from source and checkout the `develop` branch. In this way, your local working copy will be on the latest commit on the `develop` branch.

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

Notice that the code snippet above installs the library in editable mode (`-e`).

To sanity check the installation run the following:

```
>>> import ampligraph
>>> ampligraph.__version__
'2.1-dev'
```

3.1.4 Support for TensorFlow 1.x

For TensorFlow 1.x-compatible AmpliGraph, use [AmpliGraph 1.x](#), whose API are available cloning the [repository](#) from GitHub and checking out the `ampligraph1/develop` branch. However, notice that the support for this version has been discontinued.

Finally, if you want to use AmpliGraph 1.x APIs on top of Tensorflow 2, refer to the backward compatibility APIs provided on Ampligraph `compat` module.

3.2 Background

For a comprehensive theoretical and hands-on overview of KGE models and hands-on AmpliGraph, check out our tutorials: [COLING-22 KGE4NLP Tutorial \(Slides + Recording + Colab Notebook\)](#) and [ECAI-20 Tutorial \(Slides + Recording + Colab Notebook\)](#).

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 [], and YAGO [SKW07]. Countless domain-specific knowledge graphs have also been published on the web, giving birth to the so-called Web of Data [BHL11].

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 []. 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 $e_{sub}, e_{pred}, e_{obj} \in \mathcal{R}^k$ 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, RotatE [SDNT19] models relations as rotations in the complex space, 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 e_{sub} translated by the embedding of the predicate e_{pred} and the embedding of the object e_{obj} , using the L_1 or L_2 norm $||\cdot||$:

$$f_{TransE} = -||e_{sub} + e_{pred} - e_{obj}||_n$$

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 divides its APIs in the five main submodules listed below:

3.3.1 Datasets

Support for loading and managing datasets.

Loaders for Custom Knowledge Graphs

These are functions to load custom knowledge graphs from disk. They load the data from the specified files and store it as a numpy array. These loaders are recommended when the datasets to load are small in size (approx 1M entities and millions of triples), i.e., as long as they can be stored in memory. In case the dataset is too big to fit in memory, use the `GraphDataLoader` class instead (see the Advanced Topics section for more).

<code>load_from_csv(directory_path, file_name[, ...])</code>	Load a knowledge graph from a .csv file.
<code>load_from_ntriples(folder_name, file_name[, ...])</code>	Load a dataset of RDF ntriples.
<code>load_from_rdf(folder_name, file_name[, ...])</code>	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 filtering duplicated statements. In the .csv file, each line has to represent a triple, and entities and relations are separated by `sep`. For instance, if `sep="\t"`, the .csv file look like:

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

Hint: To split a generic knowledge graphs into **training**, **validation**, and **test** sets do not use the above function, but rather `train_test_split_no_unseen()`: this will return validation and test sets not including triples with entities not present 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: "\t").
- **header** (`int or 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)

Example

```
>>> PATH_TO_FOLDER = 'your/path/to/folder/'
>>> from ampligraph.datasets import load_from_csv
>>> X = load_from_csv(PATH_TO_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 a dataset of 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 .
```

Hint: To split a generic knowledge graphs into **training**, **validation**, and **test** sets do not use the above function, but rather `train_test_split_no_unseen()`: this will return validation and test sets not including triples with entities not present 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. This function, indeed, is faster by orders of magnitude.

Hint: To split a generic knowledge graphs into **training**, **validation**, and **test** sets do not use the above function, but rather `train_test_split_no_unseen()`: this will return validation and test sets not including triples with entities not present 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 $\langle s, p, o \rangle$ in the dataset this creates a corresponding triple with reciprocal relation $\langle o, p_reciprocal, s \rangle$ (default: *False*).

Returns

triples – The actual triples of the file.

Return type

ndarray, shape (n, 3)

Benchmark Datasets Loaders

The following helper functions allow to load datasets used in graph representation learning literature as benchmarks. Among the various datasets, some include additional content to the usual triples. *WN11* and *FB13* provide true and negative labels for the triples in the validation and tests sets. *CODEX-M* contains also ground truths negative triples for test and validation sets (more information about the dataset in []).

Finally, even though some of them are nowadays deprecated (*WN18* and *FB15k*), they are kept in the library as these were the first benchmarks to be used in literature.

<code>load_fb15k_237([check_md5hash, ...])</code>	Load the FB15k-237 dataset (with option to load human labeled test subset).
<code>load_wn18rr([check_md5hash, clean_unseen, ...])</code>	Load the WN18RR dataset.
<code>load_yago3_10([check_md5hash, clean_unseen, ...])</code>	Load the YAGO3-10 dataset.
<code>load_wn11([check_md5hash, clean_unseen, ...])</code>	Load the WordNet11 (WN11) dataset.
<code>load_fb13([check_md5hash, clean_unseen, ...])</code>	Load the Freebase13 (FB13) dataset.
<code>load_codecx([check_md5hash, clean_unseen, ...])</code>	Load the CoDEx-M dataset.
<code>load_fb15k([check_md5hash, add_reciprocal_rels])</code>	Load the FB15k dataset.
<code>load_wn18([check_md5hash, add_reciprocal_rels])</code>	Load the WN18 dataset.

load_fb15k_237

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

Load the FB15k-237 dataset (with option to load human labeled test subset).

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

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

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 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*: 272,115 triples
- *valid*: 17,535 triples
- *test*: 20,466 triples

It also contains a subset of the test set with human-readable labels, available here:

- *test-human*
- *test-human-ids*

Dataset	Train	Valid	Test	Test-Human	Entities	Relations
FB15K-237	272,115	17,535	20,466	273	14,541	237

Parameters

- `check_md5hash (bool)` – If *True* check the md5hash of the files (default: *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*).
- `return_mapper (bool)` – Whether to return human-readable labels in a form of dictionary in `X["mapper"]` field (default: *False*).

Returns

`splits` – The dataset splits: `{'train': train, 'valid': valid, 'test': test, 'test-human':test_human, 'test-human-ids': test_human_ids}`. Each split is a ndarray of shape (n, 3).

Return type

`dict`

Example

```
>>> 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].

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

The WN18RR dataset is loaded from file if it exists at the `AMPLIGRAPH_DATA_HOME` location. If `AMPLIGRAPH_DATA_HOME` is not set, 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`.

This dataset is divided in three splits:

- *train*: 86,835 triples
- *valid*: 3,034 triples
- *test*: 3,134 triples

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

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 dataset files (default: *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 a ndarray of shape (n, 3).

Return type

dict

Example

```
>>> from ampligraph.datasets import load_wn18rr
>>> X = load_wn18rr()
>>> X["valid"][@]
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 [], 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 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`.

This dataset is divided in three splits:

- *train*: 1,079,040 triples
- *valid*: 5,000 triples
- *test*: 5,000 triples

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

Parameters

- `check_md5hash (bool)` – If *True* check the md5hash of the files (default: *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 a ndarray of shape (n, 3).

Return type

dict

Example

```
>>> from ampligraph.datasets import load_yago3_10  
>>> X = load_yago3_10()  
>>> X["valid"][0]  
array(['Mikheil_Khutsishvili', 'playsFor', 'FC_Merani_Tbilisi'], 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* [1].

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

WN11 dataset is loaded from file if it exists at the `AMPLIGRAPH_DATA_HOME` location. If `AMPLIGRAPH_DATA_HOME` is not set, 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`.

This dataset is divided in three splits:

- `train`: 110361 triples
- `valid`: 5215 triples
- `test`: 21035 triples

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 (bool)` – If `True` check the md5hash of the files (default: `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 a ndarray of shape (n, 3). The labels are a ndarray of shape (n).

Return type

dict

Example

```
>>> from ampligraph.datasets import load_wn11
>>> X = load_wn11()
>>> X[\"valid\"][:3]
array(['__genus_xylomelum_1', '_type_of', '__dicot_genus_1'], dtype=object)
>>> X[\"valid_labels\"][: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 [] and was initially presented in *Reasoning With Neural Tensor Networks for Knowledge Base Completion* [].

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

FB13 dataset is loaded from file if it exists at the `AMPLIGRAPH_DATA_HOME` location. If `AMPLIGRAPH_DATA_HOME` is not set, 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`.

This dataset is divided in three splits:

- *train*: 316232 triples
- *valid*: 11816 triples
- *test*: 47464 triples

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 (bool)` – If *True* check the md5hash of the files (default: *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 a ndarray of shape (n, 3). The labels are ndarray of shape (n).

Return type

dict

Example

```
>>> 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)
```

load_codecx

`ampligraph.datasets.load_codecx(check_md5hash=False, clean_unseen=True, add_reciprocal_rels=False, return_mapper=False)`

Load the CoDEx-M dataset.

The dataset is described in [].

Note: CODEX-M contains also ground truths negative triples for test and validation sets. For more information, see the above reference to the original paper.

The CodDEx dataset is loaded from file if it exists at the `AMPLIGRAPH_DATA_HOME` location. If `AMPLIGRAPH_DATA_HOME` is not set, 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`.

This dataset is divided in three splits:

- *train*: 185,584 triples
- *valid*: 10,310 triples
- *test*: 10,310 triples

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	Valid-negatives	Test	Test-negatives	Entities	Relations
CoDEx-M	185,584	10,310	10,310	10311	10311	17,050	51

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 (default: *False*).

- **add_reciprocal_rels** (*bool*) – Flag which specifies whether to add reciprocal relations. For every $\langle s, p, o \rangle$ in the dataset this creates a corresponding triple with reciprocal relation $\langle o, p_{\text{reciprocal}}, s \rangle$ (default: *False*).
- **return_mapper** (*bool*) – Whether to return human-readable labels in a form of dictionary in $X["\text{mapper}"]$ field (default: *False*).

Returns

`splits` – The dataset splits: $\{\text{'train': } train, \text{'valid': } valid, \text{'valid_negatives': } valid_negatives, \text{'test': } test, \text{'test_negatives': } test_negatives\}$. Each split is a ndarray of shape (n, 3).

Return type

dict

Example

```
>>> from ampligraph.datasets import load_codex
>>> X = load_codex()
>>> X["valid"][@]
array(['Q60684', 'P106', 'Q4964182'], dtype=object)
>>> X = load_codex(return_mapper=True)
>>> [X['mapper'][elem]['label'] for elem in X['valid'][@]]
['Novalis', 'occupation', 'philosopher']
```

load_fb15k

`ampligraph.datasets.load_fb15k(check_md5hash=False, add_reciprocal_rels=False)`

Load the FB15k dataset.

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

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.

The FB15k dataset is loaded from file if it exists at the `AMPLIGRAPH_DATA_HOME` location. If `AMPLIGRAPH_DATA_HOME` is not set, 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*: 483,142 triples
- *valid*: 50,000 triples
- *test*: 59,071 triples

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

Parameters

- **check_md5hash** (*bool*) – If *True* check the md5hash of the files (default: *False*).

- **add_reciprocal_rels (bool)** – Flag which specifies whether to add reciprocal relations. For every $\langle s, p, o \rangle$ in the dataset this creates a corresponding triple with reciprocal relation $\langle o, p_reciprocal, s \rangle$ (default: *False*).

Returns

splits – The dataset splits: {‘train’: *train*, ‘valid’: *valid*, ‘test’: *test*}. Each split is a ndarray of shape (n, 3).

Return type

dict

Example

```
>>> 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.

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

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.

The WN18 dataset is loaded from file if it exists at the `AMPLIGRAPH_DATA_HOME` location. If `AMPLIGRAPH_DATA_HOME` is not set, 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 (default: *False*).

- **add_reciprocal_rels** (*bool*) – Flag which specifies whether to add reciprocal relations. For every $\langle s, p, o \rangle$ in the dataset this creates a corresponding triple with reciprocal relation $\langle o, p_reciprocal, s \rangle$ (default: *False*).

Returns

splits – The dataset splits {‘train’: *train*, ‘valid’: *valid*, ‘test’: *test*}. Each split is a ndarray of shape (n, 3).

Return type

dict

Example

```
>>> 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)
```

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
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
CODEX-M	185,584	10,310	10,311	17,050	51
FB15K	483,142	50,000	59,071	14,951	1,345
WN18	141,442	5,000	5,000	40,943	18

Hint: 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 so, it will search this location for the required dataset. If not, the dataset will be downloaded and placed in this directory.

If `AMPLIGRAPH_DATA_HOME` is not set, the datasets will be saved in the `~/ampligraph_datasets` directory.

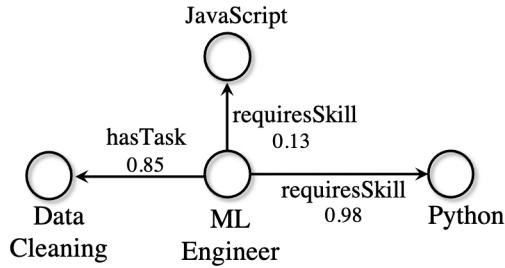
Warning:

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.

Benchmark Datasets With Numeric Values on Edges Loader

These helper functions load benchmark datasets **with numeric values on edges** (the figure below shows a toy example). Numeric values associated to edges of a knowledge graph have been used to represent uncertainty, edge importance, and even out-of-band knowledge in a growing number of scenarios, ranging from genetic data to social networks.



Hint: To process a knowledge graphs with numeric values associated to edges, enable the FocusE layer [PC21] when training a knowledge graph embedding model.

The functions will **automatically download** the datasets if they are not present in `~/ampligraph_datasets` or at the location set in the `AMPLIGRAPH_DATA_HOME`.

<code>load_onet20k([check_md5hash, clean_unseen, ...])</code>	Load the O*NET20K dataset.
<code>load_ppi5k([check_md5hash, clean_unseen, ...])</code>	Load the PPI5K dataset.
<code>load_nl27k([check_md5hash, clean_unseen, ...])</code>	Load the NL27K dataset.
<code>load_cn15k([check_md5hash, clean_unseen, ...])</code>	Load the CN15K dataset.

load_onet20k

```
ampligraph.datasets.load_onet20k(check_md5hash=False, clean_unseen=True,
                                  split_test_into_top_bottom=True, split_threshold=0.1)
```

Load the O*NET20K dataset.

O*NET20K was originally proposed in [PC21]. It is a subset of O*NET, a dataset that includes job descriptions, skills and labeled, binary relations between such concepts. Each triple is labeled with a numeric value that indicates the importance of that link.

O*NET20K dataset is loaded from file if it exists at the `AMPLIGRAPH_DATA_HOME` location. If `AMPLIGRAPH_DATA_HOME` is not set, 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`.

This dataset is divided in three splits:

- *train*: 461,932 triples
- *valid*: 850 triples
- *test*: 2,000 triples

Each triple in these splits is associated to a numeric value which represents the importance/relevance of the link.

Dataset	Train	Valid	Test	Entities	Relations
ONET*20K	461,932	850	2,000	20,643	19

Parameters

- **check_md5hash (bool)** – If *True* check the md5hash of the files (default: *False*).
- **clean_unseen (bool)** – If *True*, filters triples in validation and test sets that include entities not present in the training set.
- **split_test_into_top_bottom (bool)** – Splits the test set by numeric values and returns *test_top_split* and *test_bottom_split* by splitting based on sorted numeric values and returning top and bottom k% triples, where k is specified by *split_threshold* argument.
- **split_threshold (float)** – Specifies the top and bottom percentage of triples to return.

Returns

splits – The dataset splits: {‘train’: *train*, ‘valid’: *valid*, ‘test’: *test*, ‘test_topk’: *test_topk*, ‘test_bottomk’: *test_bottomk*, ‘train_numeric_values’: *train_numeric_values*, ‘valid_numeric_values’: *valid_numeric_values*, ‘test_numeric_values’: *test_numeric_values*, ‘test_topk_numeric_values’: *test_topk_numeric_values*, ‘test_bottomk_numeric_values’: *test_bottomk_numeric_values*}. Each *_numeric_values split contains numeric values associated to the corresponding dataset split and is a ndarray of shape (n). Each dataset split is a ndarray of shape (n,3). The *_topk and *_bottomk splits are only returned when *split_test_into_top_bottom=True* and contain the triples ordered by highest/lowest numeric edge value associated. These are typically used at evaluation time aiming at observing a model that assigns the highest rank possible to the _topk and the lowest possible to the _bottomk.

Return type

dict

Example

```
>>> from ampligraph.datasets import load_onet20k
>>> X = load_onet20k()
>>> X[“train”][0]
[‘Job_27-1021.00’ ‘has_ability_LV’ ‘1.A.1.b.2’]
>>> X[‘train_numeric_values’][0]
[0.6257143]
```

load_ppi5k

```
ampligraph.datasets.load_ppi5k(check_md5hash=False, clean_unseen=True,
                                split_test_into_top_bottom=True, split_threshold=0.1)
```

Load the PPI5K dataset.

Originally proposed in [], PPI5K is a subset of the protein-protein interactions (PPI) knowledge graph []. Numeric values represent the confidence of the link based on existing scientific literature evidence.

PPI5K is loaded from file if it exists at the AMPLIGRAPH_DATA_HOME location. If AMPLIGRAPH_DATA_HOME is not set, 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 into three splits:

- *train*: 230,929 triples
- *valid*: 19,017 triples
- *test*: 21,720 triples

Each triple in these splits is associated to a numeric value which models additional information on the fact (importance, relevance of the link).

Dataset	Train	Valid	Test	Entities	Relations
PPI5K	230929	19017	21720	4999	7

Parameters

- **check_md5hash (bool)** – If *True* check the md5hash of the files (default: *False*).
- **clean_unseen (bool)** – If *True*, filters triples in validation and test sets that include entities not present in the training set.
- **split_test_into_top_bottom (bool)** – When set to *True*, the function also returns subsets of the test set that includes only the top-k or bottom-k numeric-enriched triples. Splits the test set by numeric values and returns *test_topk_split* and *test_bottomk_split* by splitting based on sorted numeric values and returning top and bottom k% triples, where k is specified by the *split_threshold* argument.
- **split_threshold (float)** – Specifies the top and bottom percentage of triples to return.

Returns

splits – The dataset splits: `{'train': train, 'valid': valid, 'test': test, 'test_topk': test_topk, 'test_bottomk': test_bottomk, 'train_numeric_values': train_numeric_values, 'valid_numeric_values': valid_numeric_values, 'test_numeric_values': test_numeric_values, 'test_topk_numeric_values': test_topk_numeric_values, 'test_bottomk_numeric_values': test_bottomk_numeric_values}`. Each `*_numeric_values` split contains numeric values associated to the corresponding dataset split and is a ndarray of shape (n). Each dataset split is a ndarray of shape (n,3). The `*_topk` and `*_bottomk` splits are only returned when `split_test_into_top_bottom=True` and contain the triples ordered by highest/lowest numeric edge value associated. These are typically used at evaluation time aiming at observing a model that assigns the highest rank possible to the `_topk` and the lowest possible to the `_bottomk`.

Return type

dict

Example

```
>>> from ampligraph.datasets import load_ppi5k
>>> X = load_ppi5k()
>>> X["train"][@]
['4001' '5' '4176']
>>> X['train_numeric_values'][@]
[0.329]
```

load_nl27k

```
ampligraph.datasets.load_nl27k(check_md5hash=False, clean_unseen=True,  
                                split_test_into_top_bottom=True, split_threshold=0.1)
```

Load the NL27K dataset.

NL27K was originally proposed in []. It is a subset of the Never Ending Language Learning (NELL) dataset [], which collects data from web pages. Numeric values on triples represent link uncertainty.

NL27K is loaded from file if it exists at the AMPLIGRAPH_DATA_HOME location. If AMPLIGRAPH_DATA_HOME is not set, 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 into three splits:

- *train*: 149,100 triples
- *valid*: 12,274 triples
- *test*: 14,026 triples

Each triple in these splits is associated to a numeric value which represents the importance/relevance of the link.

Dataset	Train	Valid	Test	Entities	Relations
NL27K	149,100	12,274	14,026	27,221	405

Parameters

- **check_md5hash** (*bool*) – If *True* check the md5hash of the files (default: *False*).
- **clean_unseen** (*bool*) – If *True*, filters triples in validation and test sets that include entities not present in the training set.
- **split_test_into_top_bottom** (*bool*) – Splits the test set by numeric values and returns *test_top_split* and *test_bottom_split* by splitting based on sorted numeric values and returning top and bottom k% triples, where k is specified by *split_threshold* argument.
- **split_threshold** (*float*) – Specifies the top and bottom percentage of triples to return.

Returns

splits – The dataset splits: {‘train’: *train*, ‘valid’: *valid*, ‘test’: *test*, ‘test_topk’: *test_topk*, ‘test_bottomk’: *test_bottomk*, ‘train_numeric_values’: *train_numeric_values*, ‘valid_numeric_values’: *valid_numeric_values*, ‘test_numeric_values’: *test_numeric_values*, ‘test_topk_numeric_values’: *test_topk_numeric_values*, ‘test_bottomk_numeric_values’: *test_bottomk_numeric_values*}. Each *_numeric_values split contains numeric values associated to the corresponding dataset split and is a ndarray of shape (n). Each dataset split is a ndarray of shape (n,3). The *_topk and *_bottomk splits are only returned when *split_test_into_top_bottom=True* and contain the triples ordered by highest/lowest numeric edge value associated. These are typically used at evaluation time aiming at observing a model that assigns the highest rank possible to the _topk and the lowest possible to the _bottomk.

Return type

dict

Example

```
>>> from ampligraph.datasets import load_nl27k
>>> X = load_nl27k()
>>> X[["train"]][0]
['concept:company:business_review' 'concept:competeswith' 'concept:company:miami_'
 'herald001']
>>> X['train_numeric_values'][0]
[0.859375]
```

load_cn15k

```
ampligraph.datasets.load_cn15k(check_md5hash=False, clean_unseen=True,
                                split_test_into_top_bottom=True, split_threshold=0.1)
```

Load the CN15K dataset.

CN15K was originally proposed in [], it is a subset of ConceptNet [], a common-sense knowledge graph built to represent general human knowledge. Numeric values on triples represent uncertainty.

CN15k dataset is loaded from file if it exists at the `AMPLIGRAPH_DATA_HOME` location. If `AMPLIGRAPH_DATA_HOME` is not set, 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 into three splits:

- *train*: 199,417 triples
- *valid*: 16,829 triples
- *test*: 19,224 triples

Each triple in these splits is associated to a numeric value which represents the importance/relevance of the link.

Dataset	Train	Valid	Test	Entities	Relations
CN15K	199,417	16,829	19,224	15,000	36

Parameters

- `check_md5hash (bool)` – If *True*, check the md5hash of the files (default: *False*).
- `clean_unseen (bool)` – If *True*, filters triples in validation and test sets that include entities not present in the training set.
- `split_test_into_top_bottom (bool)` – Splits the test set by numeric values and returns `test_top_split` and `test_bottom_split` by splitting based on sorted numeric values and returning top and bottom k% triples, where k is specified by `split_threshold` argument.
- `split_threshold (float)` – Specifies the top and bottom percentage of triples to return.

Returns

`splits` – The dataset splits: `{'train': train, 'valid': valid, 'test': test, 'test_topk': test_topk, 'test_bottomk': test_bottomk, 'train_numeric_values': train_numeric_values, 'valid_numeric_values': valid_numeric_values, 'test_numeric_values': test_numeric_values, 'test_topk_numeric_values': test_topk_numeric_values, 'test_bottomk_numeric_values': test_bottomk_numeric_values}`. Each `*_numeric_values` split contains numeric values associated to the corresponding dataset split and is a ndarray of shape (n). Each dataset split is a ndarray of shape (n,3). The `*_topk` and `*_bottomk` splits are only returned when

`split_test_into_top_bottom=True` and contain the triples ordered by highest/lowest numeric edge value associated. These are typically used at evaluation time aiming at observing a model that assigns the highest rank possible to the `_topk` and the lowest possible to the `_bottomk`.

Return type

dict

Example

```
>>> from ampligraph.datasets import load_cn15k
>>> X = load_cn15k()
>>> X["train"][@]
['260' '2' '13895']
>>> X['train_numeric_values'][@]
[0.8927088]
```

Datasets Summary

Dataset	Train	Valid	Test	Entities	Relations
O*NET20K	461,932	138	2,000	20,643	19
PPI5K	230,929	19,017	21,720	4,999	7
NL27K	149,100	12,274	14,026	27,221	405
CN15K	199,417	16,829	19,224	15,000	36

3.3.2 Models

This module includes neural graph embedding models and support functions.

Knowledge graph embedding models are neural architectures that encode concepts from a knowledge graph (i.e., entities \mathcal{E} and relation types \mathcal{R}) into low-dimensional, continuous vectors $\in \mathbb{R}^k$. Such *knowledge graph embeddings* have applications in knowledge graph completion, entity resolution, and link-based clustering, just to cite a few [1].

Knowledge Graph Embedding models (KGE) 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 living in \mathbb{R}^k , where k can be specified by the user.

Knowledge Graph Embeddings have applications in knowledge graph completion, entity resolution, and link-based clustering, just to cite a few [2].

In Ampligraph 2, KGE models are implemented in the `ScoringBasedEmbeddingModel` class, that inherits from `Keras Model`:

<code>ScoringBasedEmbeddingModel(*args, **kwargs)</code>	Class for handling KGE models which follows the ranking based protocol.
--	---

The advantage of inheriting from Keras models are many. We can use most of Keras initializers (HeNormal, GlorotNormal...), regularizers (L^1 , L^2 ...), optimizers (Adam, AdaGrad...) and callbacks (early stopping, model checkpointing...), all without having to reimplement them. From a user perspective, people already acquainted to Keras can seemlessly work with AmpliGraph due to the similarity of the APIs.

We also provide backward compatibility with the APIs of Ampligraph 1, by wrapping the older APIs around the newer ones.

Anatomy of a Model

Knowledge Graph Embeddings are learned by training a neural architecture over a graph. Although such architecture can be of many different kinds, the training phase always consists in minimizing a *loss function* \mathcal{L} that optimizes the scores output by a *scoring function* $f_m(t)$, i.e., a model-specific function that assigns a score to a triple $t = (\text{sub}, \text{pred}, \text{obj})$.

- *Embedding Generation Layer*
- *Negatives Generation Layer*
- *Scoring Layer*
- *Loss function*
- *Optimizer*
- *Regularizer*
- *Initializer*

The first three elements are included in the `ScoringBasedEmbeddingModel` class and they inherit from `Keras Layer`.

Further, for the scoring layer and the loss function, AmpliGraph features abstract classes that can be extended to design new models:

<code>AbstractScoringLayer(*args, **kwargs)</code>	Abstract class for scoring layer.
<code>Loss([hyperparam_dict, verbose])</code>	Abstract class for the loss function.

Embedding Generation Layer

The embedding generation layer generates the embeddings of the concepts present in the triples. It may be as simple as a *shallow* encoding (i.e., a lookup of the embedding of an input node or edge type), or it can be as complex as a neural network, which tokenizes nodes and generates embeddings for nodes using a neural encoder (e.g., `NodePiece`). Currently, AmpliGraph implements the shallow look-up strategy but will be expanded soon to include other efficient approaches.

<code>EmbeddingLookupLayer(*args, **kwargs)</code>	
--	--

Negatives Generation Layer

This layer is responsible for generation of synthetic negatives. The strategies to generate negatives can be multiple. In our case, we assume a local close world assumption, and implement a simple negative generation strategy, where we randomly corrupt either the subject, the object or both the subject and the object of a triple, to generate a synthetic negative. Further, we allow filtering the true positives out of the generated negatives.

<code>CorruptionGenerationLayerTrain(*args, **kwargs)</code>	Generates corruptions during training.
--	--

Scoring Layer

The scoring layer applies a scoring function f to a triple $t = (s, p, o)$. This function combines the embeddings $\mathbf{e}_s, \mathbf{r}_p, \mathbf{e}_o \in \mathbb{R}^k$ (or $\in \mathbb{C}^k$) of the subject, predicate, and object of t into a score representing the plausibility of the triple.

<code>TransE(*args, **kwargs)</code>	Translating Embeddings (TransE) scoring layer.
<code>DistMult(*args, **kwargs)</code>	DistMult scoring layer.
<code>ComplEx(*args, **kwargs)</code>	Complex Embeddings (ComplEx) scoring layer.
<code>RotatE(*args, **kwargs)</code>	Rotate Embeddings (RotatE) scoring layer.
<code>Hole(*args, **kwargs)</code>	Holographic Embeddings (Hole) scoring layer.

Different scoring functions are designed 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 $\|\cdot\|$:

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

- **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:

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

- **RotatE** [SDNT19] models relations as rotations in the Complex space:

$$f_{RotatE} = \|\mathbf{e}_s \circ \mathbf{r}_p - \mathbf{e}_o\|$$

- **Hole** [NRP+16] uses circular correlation (denoted by \otimes):

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

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 at the compilation stage as the `loss` parameter to the `compile()` method. Below are the loss functions available in AmpliGraph.

<code>PairwiseLoss([loss_params, verbose])</code>	Pairwise, max-margin loss.
<code>AbsoluteMarginLoss([loss_params, verbose])</code>	Absolute margin, max-margin loss.
<code>SelfAdversarialLoss([loss_params, verbose])</code>	Self Adversarial Sampling loss.
<code>NLLLoss([loss_params, verbose])</code>	Negative Log-Likelihood loss.
<code>NLLMulticlass([loss_params, verbose])</code>	Multiclass Negative Log-Likelihood loss.

Regularizers

AmpliGraph includes a number of regularizers that can be used with the *loss function*. Regularizers can be passed to the `entity_relation_regularizer` parameter of `compile()` method.

`LP_regularizer()` supports L^1 , L^2 and L^3 regularization. Ampligraph also supports the `regularizers` available in TensorFlow.

<code>LP_regularizer(trainable_param[, ...])</code>	Norm L^p regularizer.
---	-------------------------

Initializers

To initialize embeddings, AmpliGraph supports all the `initializers` available in TensorFlow. Initializers can be passed to the `entity_relation_initializer` parameter of `compile()` method.

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 `optimizers` available in TensorFlow. They can be specified as the `optimizer` argument of the `compile()` method.

Training

The training procedure follows that of Keras models:

- The model is initialised as an instance of the `ScoringBasedEmbeddingModel` class. During its initialisation, we can specify, among the other hyper-parameters of the model: the size of the embedding (argument `k`); the scoring function applied by the model (argument `scoring_type`); the number of synthetic negatives generated for each triple in the training set (argument `eta`).
- The model needs to be compiled through the `compile()` method. At this stage we define, among the others, the optimizer and the objective functions. These are passed as arguments to the aforementioned method.
- The model is fitted to the training data using the `fit()` method. Next to the usual parameters that can be specified at this stage, AmpliGraph allows to also specify:
 - A `validation_filter` that contains the true positives to be removed from the synthetically corrupted triples used during validation.
 - A `focusE` option, which enables the FocusE layer [PC21]: this allows to handle datasets with a numeric value associated to the edges, which can signify importance, uncertainty, significance, confidence...
 - A `partitioning_k` argument that specifies whether the data needs to be partitioned in order to make training with datasets not fitting in memory more efficient.

For more details and options, check the `fit()` method.

Calibration

Another important feature implemented in AmpliGraph is calibration [TC20]. Such a method leverages a heuristics that significantly enhance the performance of the models. Further, it bounds the score of the model in the range $[0, 1]$, making the score of the prediction more meaningful and interpretable.

<code>CalibrationLayer(*args, **kwargs)</code>	Layer to calibrate the model outputs.
--	---------------------------------------

Numeric Values on Edges

Numeric values associated to edges of a knowledge graph have been used to represent uncertainty, edge importance, and even out-of-band knowledge in a growing number of scenarios, ranging from genetic data to social networks. Nevertheless, traditional KGE models (TransE, DistMult, ComplEx, RotatE, HolE) are not designed to capture such information, to the detriment of predictive power.

AmpliGraph includes FocusE [PC21], a method to inject numeric edge attributes into the scoring layer of a traditional KGE architecture, thus accounting for the precious information embedded in the edge weights. In order to add the FocusE layer, set `focusE=True` and specify the hyperparameters dictionary `focusE_params` in the `fit()` method.

It is possible to load some benchmark knowledge graphs with numeric-enriched edges through Ampligraph `dataset loaders`.

Saving/Restoring Models

The weights of a trained model can be saved and restored from disk. This is useful to avoid re-training a model. In order to save and restore the weights of a model, we can use the `save_weights()` and `load_weights()` methods. When the model is saved and loaded with these methods, however, it is not possible to restart the training from where it stopped. AmpliGraph gives the possibility of doing that using `save_model()` and `restore_model()` available in the `utils` module.

Compatibility Ampligraph 1.x

Provides backward compatibility to AmpliGraph 1 APIs.

For those familiar with versions of AmpliGraph 1.x, we have created backward compatible APIs under the `ampligraph.compat` module.

These APIs act as wrappers around the newer Keras style APIs and provide seamless experience for our existing user base.

The first group of APIs defines the classes that wraps around the ScoringBasedEmbeddingModel with a specific scoring function.

<code>TransE([k, eta, epochs, batches_count, ...])</code>	Class wrapping around the ScoringBasedEmbedding-Model with the TransE scoring function.
<code>ComplEx([k, eta, epochs, batches_count, ...])</code>	Class wrapping around the ScoringBasedEmbedding-Model with the ComplEx scoring function.
<code>DistMult([k, eta, epochs, batches_count, ...])</code>	Class wrapping around the ScoringBasedEmbedding-Model with the DistMult scoring function.
<code>HolE([k, eta, epochs, batches_count, seed, ...])</code>	Class wrapping around the ScoringBasedEmbedding-Model with the HolE scoring function.

When it comes to evaluation, on the other hand, the following API wraps around the new evaluation process of AmpliGraph 2.

<code>evaluate_performance(X, model[, ...])</code>	Evaluate the performance of an embedding model.
--	---

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.

After the training is complete, the model is ready to perform predictions and to be evaluated on unseen data. Given a triple, the model can score it and quantify its plausibility. Importantly, the entities and relations of new triples must have been seen during training, otherwise no embedding for them is available. Future extensions of the code base will introduce inductive methods as well.

The standard evaluation of a test triples is achieved by comparing the score assigned by the model to that triple with those assigned to the same triple where we corrupted either the object or the subject. From this comparison we extract some metrics. By aggregating the metrics obtained for all triples in the test set, we finally obtain a “thorough” (depending on the quality of the test set and of the corruptions) evaluation of the model.

Metrics

The available metrics implemented in AmpliGraph to rank a triple against its corruptions are listed in the table below.

<code>rank_score(y_true, y_pred[, pos_lab])</code>	Computes the rank of a triple.
<code>mr_score(ranks)</code>	Mean Rank (MR).
<code>mrr_score(ranks)</code>	Mean Reciprocal Rank (MRR).
<code>hits_at_n_score(ranks, n)</code>	Hits@N.

Model Selection

AmpliGraph implements a model selection routine for KGE models via either a grid search or a random search. Random search is typically more efficient, but grid search, on the other hand, can provide a more controlled selection framework.

<code>select_best_model_ranking(model_class, ...)</code>	Model selection routine for embedding models via either grid search or random search.
--	---

Helper Functions

Utilities and support functions for evaluation procedures.

<code>train_test_split_no_unseen(X[, test_size, ...])</code>	Split into train and test sets.
<code>filter_unseen_entities(X, model[, verbose])</code>	Filter unseen entities in the test set.

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.

<code>discover_facts(X, model[, top_n, strategy, ...])</code>	Discover new facts from an existing knowledge graph.
<code>find_clusters(X, model[, ...])</code>	Perform link-based cluster analysis on a knowledge graph.
<code>find_duplicates(X, model[, mode, metric, ...])</code>	Find duplicate entities, relations or triples in a graph based on their embeddings.
<code>query_topn(model[, top_n, head, relation, ...])</code>	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.latent_features.ScoringBasedEmbeddingModel.evaluate()` method. 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 entities and relations or 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 have not 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 of 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** (*str*) – The candidates generation strategy:
 - **'random_uniform'**
[generates N candidates ($N \leq max_candidates$) 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)

Example

```
>>> import requests
>>> from ampligraph.latent_features import ScoringBasedEmbeddingModel
>>> from ampligraph.datasets import load_from_csv
>>> 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)
>>> dataset = load_from_csv('.', 'GoT.csv', sep=',')
>>> model = ScoringBasedEmbeddingModel(eta=5,
>>>                                     k=300,
>>>                                     scoring_type='ComplEx')
>>> model.compile(optimizer='adam', loss='multiclass_nll')
>>> model.fit(dataset,
```

(continues on next page)

(continued from previous page)

```

>>>         batch_size=100,
>>>         epochs=10,
>>>         validation_freq=50,
>>>         validation_batch_size=100,
>>>         validation_data = dataset['valid'])
>>> discover_facts(dataset,
>>>                  model,
>>>                  top_n=100,
>>>                  strategy='random_uniform',
>>>                  max_candidates=100,
>>>                  target_rel='ALLIED_WITH',
>>>                  seed=0)
Epoch 1/10
33/33 [=====] - 1s 27ms/step - loss: 177.7778
Epoch 2/10
33/33 [=====] - 0s 6ms/step - loss: 177.4795
Epoch 3/10
33/33 [=====] - 0s 6ms/step - loss: 176.9654
Epoch 4/10
33/33 [=====] - 0s 6ms/step - loss: 175.8453
Epoch 5/10
33/33 [=====] - 0s 6ms/step - loss: 173.4385
Epoch 6/10
33/33 [=====] - 0s 6ms/step - loss: 168.8143
Epoch 7/10
33/33 [=====] - 0s 6ms/step - loss: 161.2919
Epoch 8/10
33/33 [=====] - 0s 6ms/step - loss: 151.3496
Epoch 9/10
33/33 [=====] - 0s 6ms/step - loss: 140.4268
Epoch 10/10
33/33 [=====] - 0s 5ms/step - loss: 129.8206
3175 triples containing invalid keys skipped!
(array([['House Nymeros Martell of Sunspear', 'ALLIED_WITH',
        'House Mallister of Seagard'],
       ['Ben', 'ALLIED_WITH', 'House Mallister of Seagard'],
       ['Selwyn Tarth', 'ALLIED_WITH', 'House Mallister of Seagard'],
       ['Clarence Charlton', 'ALLIED_WITH', 'House Woods'],
       ['Selwyn Tarth', 'ALLIED_WITH', 'House Woods'],
       ['Dacks', 'ALLIED_WITH', 'Titus Peake'],
       ['Barra', 'ALLIED_WITH', 'Titus Peake'],
       ['House Chelsted', 'ALLIED_WITH', 'Denys Darklyn'],
       ['Crow Spike Keep', 'ALLIED_WITH', 'Denys Darklyn'],
       ['Selwyn Tarth', 'ALLIED_WITH', 'Denys Darklyn'],
       ['House Chelsted', 'ALLIED_WITH', 'House Belmore of Strongsong'],
       ['Barra', 'ALLIED_WITH', 'House Belmore of Strongsong'],
       ['Walder Frey', 'ALLIED_WITH', 'House Belmore of Strongsong']],
      dtype=object),
array([ 2. , 53. , 73. , 42. , 18. , 59.5, 86. , 76.5, 31. , 60.5, 31.5,
       32. , 24. ]))

```

find_clusters

`ampligraph.discovery.find_clusters(X, model, clustering_algorithm=DBSCAN(), mode='e')`

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 to be 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** (*str*) – Clustering mode.

Choose from:

- '`e`' (default): the algorithm will cluster the embeddings of the provided entities.
- '`r`': the algorithm will cluster the embeddings of the provided relations.
- '`t`': 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]`

Example

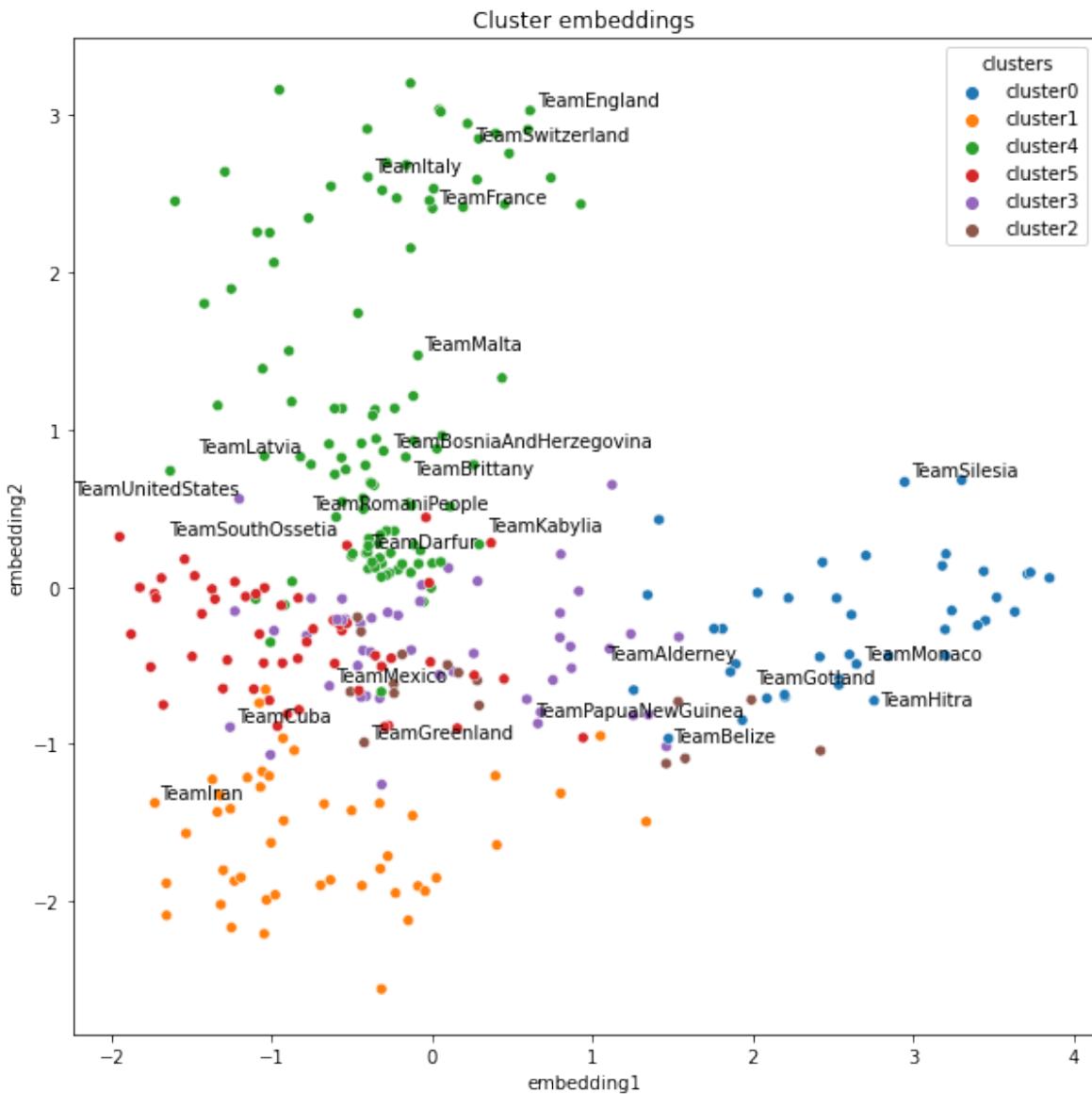
```
>>> # Note seaborn, matplotlib, adjustText are not AmpliGraph dependencies.  
>>> # and must therefore be installed manually as:  
>>> #  
>>> # $ pip install seaborn matplotlib adjustText  
>>>  
>>> import requests  
>>> import pandas as pd  
>>> import numpy as np  
>>> from sklearn.decomposition 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 ScoringBasedEmbeddingModel  
>>> 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 = ScoringBasedEmbeddingModel(eta=5,  
>>>                                     k=300,  
>>>                                     scoring_type='ComplEx')  
>>> model.compile(optimizer='adam', loss='multiclass_nll')  
>>> model.fit(X,  
>>>             batch_size=10000,  
>>>             epochs=10)  
>>>  
>>> 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='e')  
>>>  
>>> 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='e')  
>>>
```

(continues on next page)

(continued from previous page)

```
>>> # 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']+0.02, point['embedding2'], u
-> str(point['teams'])))
>>> adjust_text(texts)
```



find_duplicates

```
ampligraph.discovery.find_duplicates(X, model, mode='e', 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 “*e*” mode to find all the movies that could be 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 to be 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** (*str*) – Specifies among which type of entities to look for duplicates.

Choose from:

- ‘e’ (default): the algorithm will find duplicates of the provided entities based on their embeddings.
- ‘r’: the algorithm will find duplicates of the provided relations based on their embeddings.
- ‘t’ : 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='auto'`. Should be between 0 and 1 (default: 0.1).
- **verbose** (*bool*) – Whether to print evaluation messages during optimisation when `tolerance='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 if the automatic tolerance option is selected).

Example

```
>>> import pandas as pd
>>> import numpy as np
>>> import re
>>> from ampligraph.latent_features.models import ScoringBasedEmbeddingModel
>>> # The IMDB dataset used here is part of the Movies5 dataset found on:
>>> # The Magellan Data Repository (https://sites.google.com/site/anhaiidgroup/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 ScoringBasedEmbeddingModel
>>>
>>> imdb_triples = np.array(imdb_triples)
>>> model = ScoringBasedEmbeddingModel(eta=5,
>>>                                     k=300,
>>>                                     scoring_type='ComplEx')
>>> model.compile(optimizer='adam', loss='multiclass_nll')
```

(continues on next page)

(continued from previous page)

```

>>> model.fit(imdb_triples,
>>>           batch_size=10000,
>>>           epochs=10)
>>>
>>> # Finding duplicates movies (entities)
>>> from ampligraph.discovery import find_duplicates
>>>
>>> entities = np.unique(imdb_triples[:, 0])
>>> dups, _ = find_duplicates(entities, model, mode='e', tolerance=0.45)
>>> id_list = []
>>> for data in dups:
>>>     for i in data:
>>>         id_list.append(int(i[2]))
>>> print(imdb.iloc[id_list[:6]][['movie_name', 'year']])
Epoch 1/10
7/7 [=====] - 1s 122ms/step - loss: 15612.8799
Epoch 2/10
7/7 [=====] - 0s 20ms/step - loss: 15610.5010
Epoch 3/10
7/7 [=====] - 0s 19ms/step - loss: 15607.7412
Epoch 4/10
7/7 [=====] - 0s 19ms/step - loss: 15604.0674
Epoch 5/10
7/7 [=====] - 0s 20ms/step - loss: 15598.9365
Epoch 6/10
7/7 [=====] - 0s 19ms/step - loss: 15591.7188
Epoch 7/10
7/7 [=====] - 0s 19ms/step - loss: 15581.6055
Epoch 8/10
7/7 [=====] - 0s 20ms/step - loss: 15567.6807
Epoch 9/10
7/7 [=====] - 0s 20ms/step - loss: 15548.8184
Epoch 10/10
7/7 [=====] - 0s 21ms/step - loss: 15523.8721
      movie_name  year
5198  Duel to Death  1983
5199  Duel to Death  1983
2649  The Eliminator 2004
2650  The Eliminator 2004
3967  Lipstick Camera 1994
3968  Lipstick Camera 1994

```

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 $\langle \text{subject}, \text{predicate} \rangle$ pair in the arguments, the model will score all possible triples $\langle \text{subject}, \text{predicate}, ? \rangle$, filling in the missing element with known entities, and return the top_n triples ordered by score. If given a $\langle \text{subject}, \text{object} \rangle$ pair it will fill in the missing element with known relations.

Therefore, if we feed the function with $\langle \text{subject}, \text{predicate} \rangle$ or $\langle \text{predicate}, \text{object} \rangle$, it solves the link prediction task.

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

Parameters

- **model** (*ScoringBasedEmbeddingModel*) – The trained model that will be used to score triple completions.
- **top_n** (*int*) – The number of completed triples to returned.
- **head** (*str*) – An entity string to query.
- **relation** (*str*) – A relation string to query.
- **tail** (*str*) – 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 of shape (n, 3)*) – A list of triples ordered by score.
- **S** (*ndarray, shape (n)*) – A list of scores.

Example

```
>>> import requests  
>>> from ampligraph.datasets import load_from_csv  
>>> from ampligraph.discovery import discover_facts  
>>> from ampligraph.discovery import query_topn  
>>> from ampligraph.latent_features import ScoringBasedEmbeddingModel  
>>> # 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 = ScoringBasedEmbeddingModel(eta=5,  
                                         k=150,
```

(continues on next page)

(continued from previous page)

```
>>> scoring_type='TransE')
>>> model.compile(optimizer='adagrad', loss='pairwise')
>>> model.fit(X,
>>>             batch_size=100,
>>>             epochs=20,
>>>             verbose=False)
>>>
>>> query_topn(model, top_n=5,
>>>              head='Eddard Stark', relation='ALLIED_WITH', tail=None,
>>>              ents_to_consider=None, rels_to_consider=None)
>>>
(array([['Eddard Stark', 'ALLIED_WITH', 'Smithyton'],
       ['Eddard Stark', 'ALLIED_WITH', 'Eden Risley'],
       ['Eddard Stark', 'ALLIED_WITH', 'House Westbrook'],
       ['Eddard Stark', 'ALLIED_WITH', 'House Leygood'],
       ['Eddard Stark', 'ALLIED_WITH', 'House Bridges']], dtype='|<U44'),
array([9.000417, 5.272001, 5.1876183, 5.121145, 5.0564814],
      dtype=float32))
```

3.3.5 Utils

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

This module contains utility functions for Knowledge Graph Embedding models.

Saving/Restoring Models

Models can be saved and restored from disk. This is useful to avoid re-training a model. On the contrary of what happens for `save_weights()` and `load_weights()`, the functions below allow to restart the model training from where it was interrupted when the model was first saved.

<code>save_model(model[, model_name_path, protocol])</code>	Save a trained model to disk.
<code>restore_model([model_name_path])</code>	Restore a trained model from disk.

save_model

`ampligraph.utils.save_model(model, model_name_path=None, protocol=5)`

Save a trained model to disk.

Example

```
>>> 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'],
```

(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)
>>> 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, HolE or RotatE.
- **model_name_path** (*str*) – The name of the model to be saved. If not specified, a default name with current datetime is selected and the model is saved to the working directory.

restore_model

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

Restore a trained model from disk.

Parameters

model_name_path (*str*) – Name of the path to the model.

Visualization

Functions to visualize embeddings.

```
create_tensorboard_visualizations(model, loc) Export embeddings to Tensorboard.
```

create_tensorboard_visualizations

```
ampligraph.utils.create_tensorboard_visualizations(model, loc, entities_subset='all', 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 the visualization of the embeddings using Tensorboard. This is generally for use with a `local Tensorboard instance`.
- A tab-separated file of embeddings named `embeddings_projector.tsv`. This is generally used to visualize embeddings by uploading to `TensorBoard Embedding Projector`.

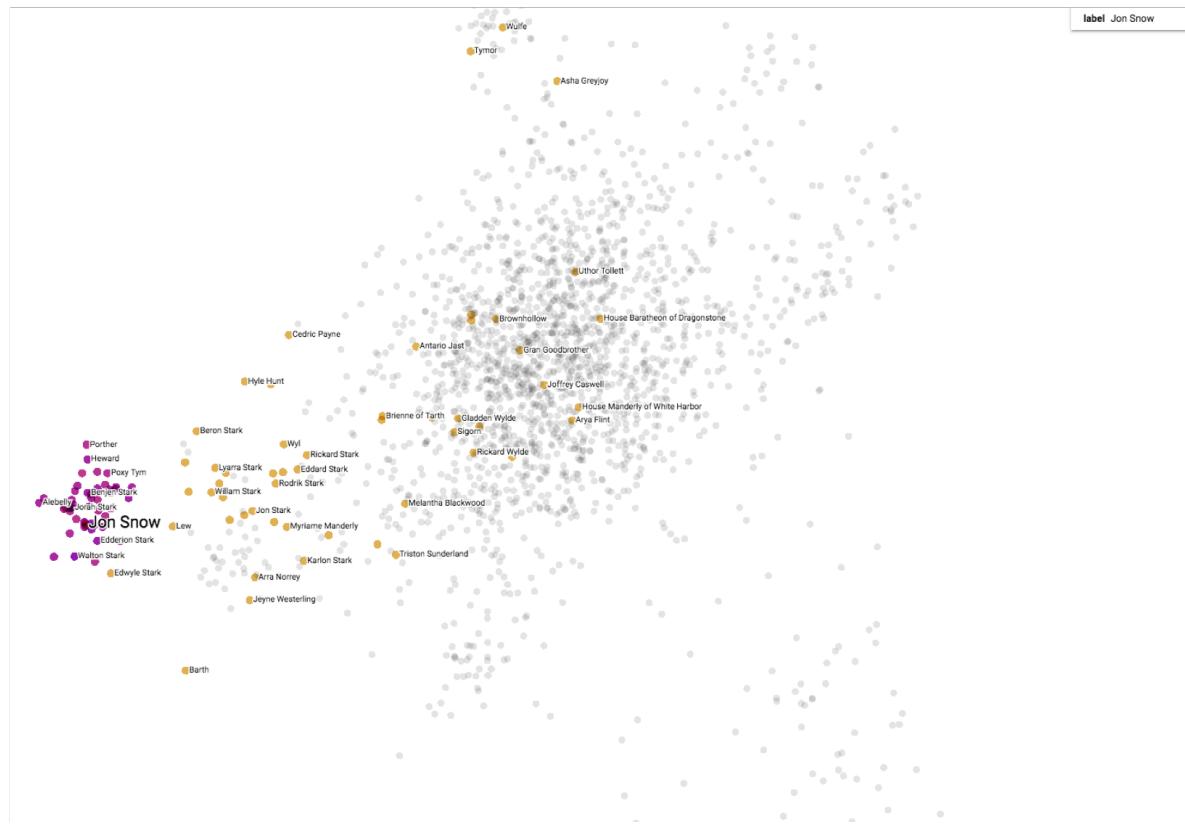
- **Embeddings metadata (i.e., the embedding labels from the original knowledge graph) saved to in a file named `metadata.tsv`**. Such file can be used in TensorBoard or uploaded to TensorBoard Embedding Projector.

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 [here](#).

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



Example

```
>>> # create model and compile using user defined optimizer settings and user-defined settings of an existing loss
>>> from ampligraph.latent_features import ScoringBasedEmbeddingModel
>>> from ampligraph.latent_features.loss_functions import SelfAdversarialLoss
>>> import tensorflow as tf
>>> optim = tf.optimizers.Adam(learning_rate=0.01)
>>> loss = SelfAdversarialLoss({'margin': 0.1, 'alpha': 5, 'reduction': 'sum'})
>>> model = ScoringBasedEmbeddingModel(eta=5,
>>>                                     k=300,
>>>                                     scoring_type='ComplEx',
>>>                                     seed=0)
>>> model.compile(optimizer=optim, loss=loss)
>>> model.fit('./fb15k-237/train.txt',
>>>           batch_size=10000,
>>>           epochs=5)
Epoch 1/5
29/29 [=====] - 2s 67ms/step - loss: 13101.9443
Epoch 2/5
29/29 [=====] - 1s 20ms/step - loss: 11907.5771
Epoch 3/5
29/29 [=====] - 1s 21ms/step - loss: 10890.3447
Epoch 4/5
29/29 [=====] - 1s 20ms/step - loss: 9520.3994
Epoch 5/5
29/29 [=====] - 1s 20ms/step - loss: 8314.7529
>>> from ampligraph.utils import create_tensorboard_visualizations
>>> create_tensorboard_visualizations(model,
>>>                                     entities_subset='all',
>>>                                     loc = './full_embeddings_vis')
>>> # On terminal run: tensorboard --logdir='./full_embeddings_vis' --port=8891
>>> # Open the browser and go to the following URL: http://127.0.0.1:8891/#projector
```

Parameters

- **model** (*EmbeddingModel*) – A trained neural knowledge graph embedding model, the model must be an instance of TransE, DistMult, ComplEx, HolE or RotatE.
- **loc** (*str*) – Directory where the files are written.
- **entities_subset** (*list*) – List of entities whose embeddings have to be visualized.
- **labels** (*pd.DataFrame*) – Label(s) for each embedding point in the Tensorboard visualization. Default behaviour is to use the embedding labels included in the model.
- **export_tsv_embeddings** (*bool*) – If *True* (default), 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*) – If *True* (default), will write a file named ‘*metadata.tsv*’ in the same directory as path.

Others

Function various functions to be used at need.

<code>dataframe_to_triples(X, schema)</code>	Convert DataFrame into triple format.
<code>preprocess_focusE_weights(data, weights[, ...])</code>	Preprocessing of focusE weights.

`dataframe_to_triples`

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

Convert DataFrame into triple format.

Parameters

- `X (pd.DataFrame with headers)` –
- `schema (list of tuples)` – List of (subject, relation_name, object) tuples where subject and object are in the headers of the data frame.

Example

```
>>> 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')
```

`preprocess_focusE_weights`

`ampligraph.utils.preprocess_focusE_weights(data, weights, normalize=True)`

Preprocessing of focusE weights.

Extract weights from data, remove *Nan*s, average weights and normalize them if `self.focusE_params['normalize_numeric_values']=True`.

Parameters

- `data (array-like, shape (n,m))` – Array of shape (n,m) with $m = 4$. If `weights=None`, data contains triples and weights ($m > 3$). If `weights` is passed, data only contains triples ($m = 3$).
- `weights (array-like)` – If not `None`, `weights` has shape (n, m-3), with $m>0$.
- `normalize (bool)` – Specify whether to normalize the weights into the [0,1] range (default: `True`).

Returns

processed_weights – An array of weights properly preprocessed and averaged into a unique vector if more than one vector of weights were given.

Return type

np.array, shape (n, 1)

3.3.6 Pre-Trained Models

Support for loading and managing pretrained models.

This module provides an API to download and have ready to use pre-trained ScoringBasedEmbeddingModel.

<code>load_pretrained_model(dataset, scoring_type)</code>	Function to load a pretrained model.
---	--------------------------------------

Currently the available models are trained on “FB15K-237”, “WN18RR”, “YAGO310”, “FB15K” and “WN18” and have as scoring function “TransE”, “DistMult”, “ComplEx”, “HolE” and “RotatE”.

The different submodules provide the user with support through all the operations needed when dealing with Knowledge Graph Embedding models, from loading benchmark or user customised datasets, to saving and reloading a model after it has been trained, validated and tested. Further, the APIs also support important downstream tasks and provide enough flexibility to allow custom extensions from the most demanding users.

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 across 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 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 .
```

3.4.5 Unit Tests

To run all the unit tests:

```
$ pytest tests
```

See [pytest documentation](#) for additional arguments.

3.4.6 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`.

3.4.7 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 ScoringBasedEmbeddingModel
from ampligraph.evaluation import mrr_score, hits_at_n_score
from ampligraph.latent_features.loss_functions import get as get_loss
from ampligraph.latent_features.regularizers import get as get_regularizer
import tensorflow as tf

# load Wordnet18 dataset:
X = load_wn18()

# Initialize a ComplEx neural embedding model: the embedding size is k,
# eta specifies the number of corruptions to generate per each positive,
# scoring_type determines the scoring function of the embedding model.
model = ScoringBasedEmbeddingModel(k=150,
                                    eta=10,
                                    scoring_type='ComplEx')

# Optimizer, loss and regularizer definition
optim = tf.keras.optimizers.Adam(learning_rate=1e-3)
loss = get_loss('pairwise', {'margin': 0.5})
regularizer = get_regularizer('LP', {'p': 2, 'lambda': 1e-5})

# Compilation of the model
model.compile(optimizer=optim, loss=loss, entity_relation_regularizer=regularizer)

# 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 = {'test' : np.concatenate((X['train'], X['valid'], X['test']))}

# Early Stopping callback
checkpoint = tf.keras.callbacks.EarlyStopping(
    monitor='val_{}'.format('hits10'),
    min_delta=0,
    patience=5,
    verbose=1,
    mode='max',
    restore_best_weights=True
)

# Fit the model on training and validation set
model.fit(X['train'],
           batch_size=int(X['train'].shape[0] / 10),
```

(continues on next page)

(continued from previous page)

```

epochs=20,           # Number of training epochs
validation_freq=20, # Epochs between successive validation
validation_burn_in=100, # Epoch to start validation
validation_data=X['valid'], # Validation data
validation_filter=filter, # Filter positives from validation corruptions
callbacks=[checkpoint], # Early stopping callback (more from tf.keras.
˓→ callbacks are supported)
verbose=True          # Enable stdout messages
)

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

# 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: 0.884418, Hits@10: 0.935500

```

3.5.2 Model selection

```

from ampligraph.datasets import load_wn18
from ampligraph.evaluation import select_best_model_ranking

# 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": [300],
    "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 adversarial loss
        "margin": [0.5, 20],
        #alpha corresponding to adversarial loss
        "alpha": [0.5]
    },
    "embedding_model_params": {

```

(continues on next page)

(continued from previous page)

```

# 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 possible combinations of hyperparameters.
# Models are validated on the validation set.
# It returns a model re-trained on training and validation sets.
best_model, best_params, best_mrr_train, \
ranks_test, test_evaluation, experimental_history = \
    select_best_model_ranking(model_class, # Name of the model to be_
→used
                                # Dataset
X_dict['train'],
X_dict['valid'],
X_dict['test'],
# Parameter grid
param_grid,
# Set maximum number of combinations
max_combinations=20,
# Use filtered set for eval
use_filter=True,
# corrupt subject and objects
→separately during eval
                                corrupt_side='s,o',
                                # Log all the model hyperparams and_
→evaluation stats
                                verbose=True)
print(type(best_model).__name__)
print("Best model parameters: ")
print(best_params)
print("Best MRR train: ", best_mrr_train)
print("Test evaluation: ", test_evaluation)
# Output:
# ComplEx

# Best model parameters:
# {'batches_count': 10, 'seed': 0, 'epochs': 300, 'k': 100, 'eta': 10,
# 'loss': 'self_adversarial', 'loss_params': {'margin': 0.5, 'alpha': 0.5},
# 'embedding_model_params': {'negative_corruption_entities': 'all'}, 'regularizer': 'LP',
# 'regularizer_params': {'p': 2, 'lambda': 0.0001}, 'optimizer': 'adam',
# 'optimizer_params': {'lr': 0.01}, 'verbose': True}

```

(continues on next page)

(continued from previous page)

```
# Best MRR train: 0.9341455440346633

# Test evaluation: {'mrr': 0.934852832005159, 'mr': 674.1877, 'hits_1': 0.9276, 'hits_3': 0.
˓→9406, 'hits_10': 0.9454}
```

3.5.3 Get the embeddings

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

model = ScoringBasedEmbeddingModel(k=5, eta=1, scoring_type='TransE')
model.compile(optimizer='adam', loss='nll')
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, epochs=5)
model.get_embeddings(['f', 'e'], embedding_type='e')
# Output
# [[ 0.5677353   0.65208733   0.66626084   0.7323714   0.43467668]
# [-0.7102897   0.59935296   0.17629518   0.5096843  -0.53681636]]
```

3.5.4 Save and Restore a Model

```
import numpy as np
from ampligraph.latent_features import ScoringBasedEmbeddingModel
from ampligraph.utils import save_model, restore_model

model = ScoringBasedEmbeddingModel(k=5, eta=1, scoring_type='ComplEx')
model.compile(optimizer='adam', loss='nll')

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, epochs=5)
```

(continues on next page)

(continued from previous page)

```
# Use the trained model to predict
y_pred_before = model.predict(np.array([[['f', 'y', 'e'], ['b', 'y', 'd']])))
print(y_pred_before)
# [ 0.1416718 -0.0070735]

# 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)
# [ 0.1416718 -0.0070735]
```

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']]
```

(continues on next page)

(continued from previous page)

```

['b' 'y' 'a']
['c' 'y' 'a']
['c' 'y' 'd']
['b' 'y' 'c']
['f' 'y' 'e']]
"""

print(X_test)

"""

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 Visualizing Embeddings

Model Training and Evaluation

```

import numpy as np
import requests
import tensorflow as tf

from ampligraph.datasets import load_from_csv

```

(continues on next page)

(continued from previous page)

```

from ampligraph.latent_features import ScoringBasedEmbeddingModel
from ampligraph.latent_features.loss_functions import get as get_loss
from ampligraph.latent_features.regularizers import get as get_regularizer
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)

# # # MODEL TRAINING # # #

# Initialize a ComplEx neural embedding model
model = ScoringBasedEmbeddingModel(k=100,
                                    eta=20,
                                    scoring_type='ComplEx')

# Optimizer, loss and regularizer definition
optim = tf.keras.optimizers.Adam(learning_rate=1e-4)
loss = get_loss('multiclass_nll')
regularizer = get_regularizer('LP', {'p': 3, 'lambda': 1e-5})

# Compilation of the model
model.compile(optimizer=optim, loss=loss, entity_relation_regularizer=regularizer)

# Fit the model
model.fit(X_train,
           batch_size=int(X_train.shape[0] / 50),
           epochs=300, # Number of training epochs
           verbose=True # Enable stdout messages
           )

# # # MODEL EVALUATION # # #
# Specify triples to filter out of corruptions since true positives
filter_triples = {'test': np.concatenate((X_train, X_test))}

# Evaluation of the model
ranks = model.evaluate(X_test,
                       use_filter=filter_triples,
                       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))

```

(continues on next page)

(continued from previous page)

```

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))
# Output:
# MRR: 0.29
# MR: 3450.72
# Hits@10: 0.41
# Hits@3: 0.34
# Hits@1: 0.22

```

Clustering and 2D Projections

Please install lib adjustText using the following command: pip install adjustText. Further, please install pycountry_convert with the following command: pip install pycountry_convert. This library is used to map countries to the corresponding continents.

```

import pandas as pd
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
import pycountry_convert as pc
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 in order to plot them
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='e')

# This function maps country to continent
def cn_to_ctn(team_name):
    try:
        country_name = ' '.join(re.findall('[A-Z][^A-Z]*', team_name[4:]))
        country_alpha2 = pc.country_name_to_country_alpha2(country_name)
        country_continent_code = pc.country_alpha2_to_continent_code(country_alpha2)
        country_continent_name = pc.convert_continent_code_to_continent_name(country_
        continent_code)
        return country_continent_name
    except KeyError:
        return "unk"

```

(continues on next page)

(continued from previous page)

```

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

    plt.savefig(hue + '_cluster_ex.png')

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

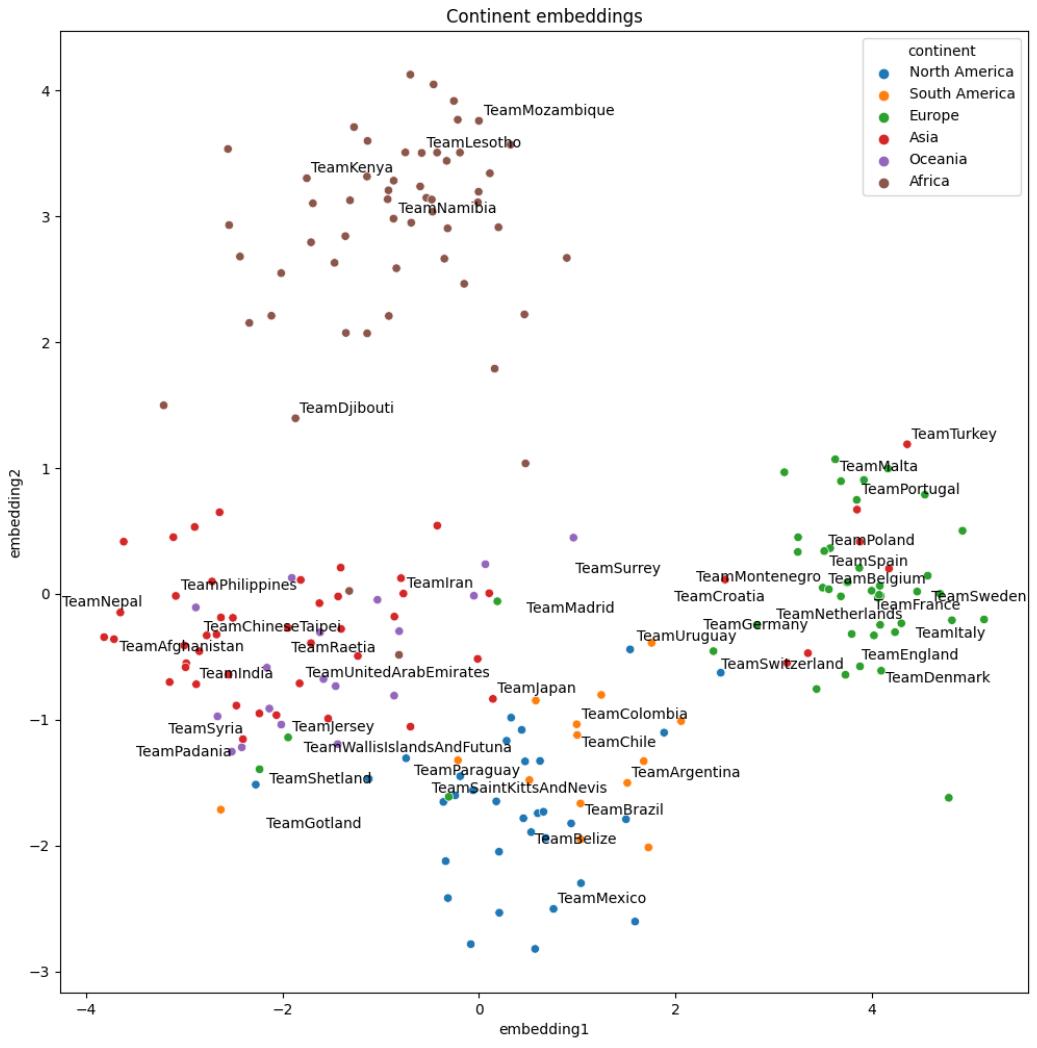
```

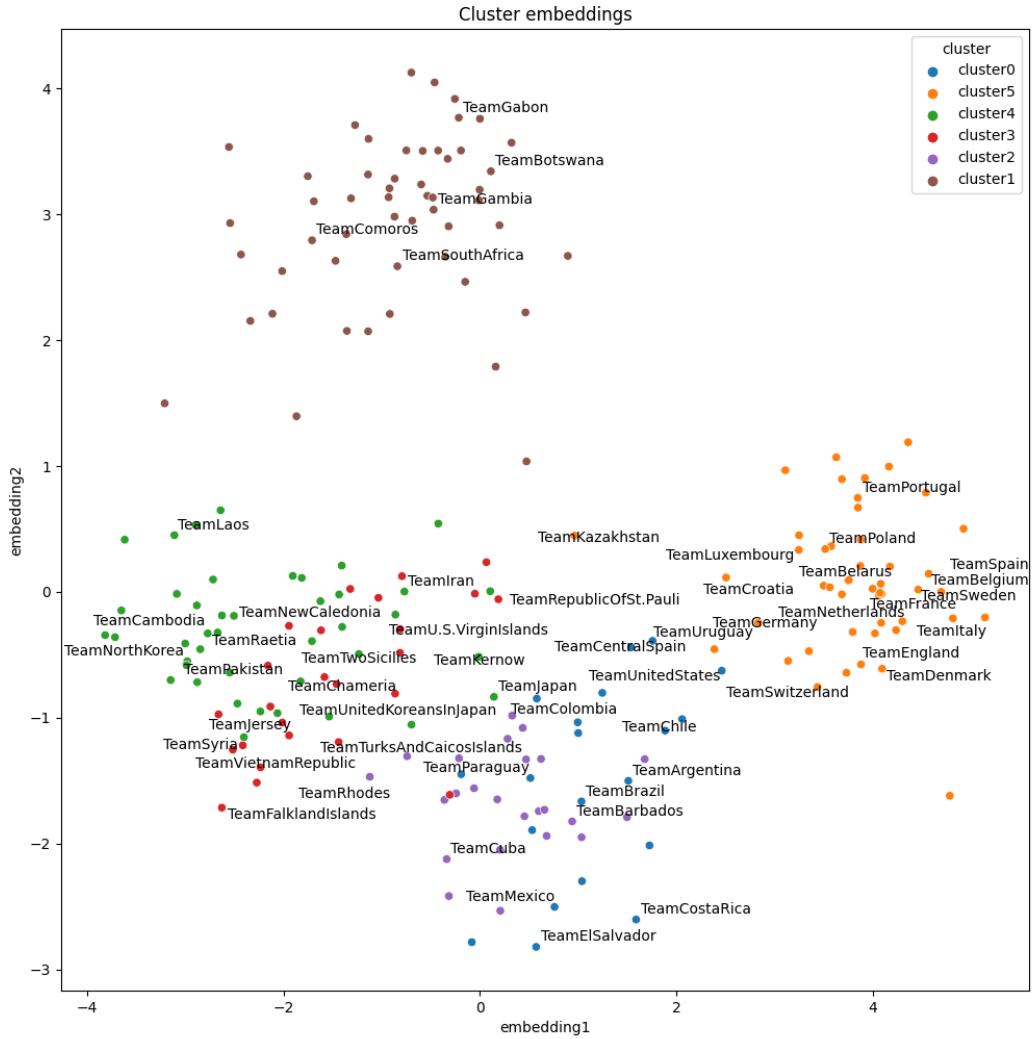
Results Visualization

```

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

```





3.6 Tutorials

For a comprehensive theoretical and hands-on overview of KGE models and hands-on AmpliGraph, check out our tutorials: [COLING-22 KGE4NLP Tutorial \(Slides + Colab Notebook\)](#) and [ECAI-20 Tutorial \(Slides + Recording + Colab Notebook\)](#).

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](#).

If you reuse materials presented in the tutorials, cite as:

```
@misc{kge4nlp_tutorial_coling22,
    title = {Knowledge Graph Embeddings for NLP: From Theory to Practice},
    url = {https://kge4nlp-coling22.github.io/},
    author= {Luca Costabello and
             Adrianna Janik and
             Eda Bayram and
             Sumit Pai},
    date = {2022-16-10},
    note = {COLING 2022 Tutorials}
}
```

```
@misc{kge_tutorial_ecai20,
    title = {Knowledge Graph Embeddings Tutorial: From Theory to Practice},
    url = {http://kge-tutorial-ecai-2020.github.io/},
    author= {Luca Costabello and
             Sumit Pai and
             Adrianna Janik and
             Nick McCarthy},
    shorttitle = {Knowledge Graph Embeddings Tutorial},
    date = {2020-09-04},
    note = {ECAI 2020 Tutorials}
}
```

3.7 Performance

3.7.1 Predictive Performance

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

Note: AmpliGraph 1.x Benchmarks. AmpliGraph 1.x predictive power report is available [here](#).

3.7.2 FB15K-237

Model	IMR	MRR	Hits@1	Hits@3	Blits@10	Hyperparameters
TransE	222	0.31	0.22	0.35	0.49	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; seed: 0; batches_count: 5;
Dist-Mult	211	0.30	0.21	0.33	0.48	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;
ComplEx	204	0.31	0.22	0.34	0.49	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: 10;
HolE	190	0.30	0.21	0.33	0.48	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;
RotatE	162	0.31	0.22	0.35	0.51	k: 350; epochs: 700; eta: 20; loss: self_adversarial; loss_params: {'margin': 5, 'alpha': 1.0}; optimizer: 'adam'; learning_rate: 1e-05; regularizer: LP; regularizer_params: {'p': 3, 'lambda': 0.001}; batches_count: 55

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	IMR	MRR	Hits@1	Hits@3	Blits@10	Hyperparameters
TransE	3143	0.22	0.03	0.38	0.52	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; embedding_model_params: norm: 1; batches_count: 150;
Dist-Mult	4832	0.47	0.43	0.48	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; batches_count: 100;
ComplEx	4356	0.51	0.47	0.52	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	7072	0.47	0.44	0.49	0.54	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;
RotatE	1839	0.52	0.47	0.53	0.61	k: 350; epochs: 350; eta: 20; loss: self_adversarial; loss_params: {'margin': 5, 'alpha': 0.5}; optimizer: adam; learning_rate: 0.000; regularizer: 'LP'; regularizer_params: {'p': 3, 'lambda': 1e-05}; batches_count: 18

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	IMR	MRR	Hits@10	Hits@50	Hits@100	Hyperparameters
TransE1210	0.50	0.41	0.56	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; seed: 0; batches_count: 100;
Dist-Mult	2301	0.48	0.39	0.53	0.64	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; batches_count: 100;
ComplEx	3153	0.49	0.40	0.54	0.65	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	6941	0.47	0.39	0.52	0.62	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
RotatE	1318	0.43	0.33	0.48	0.63	k: 350; epochs: 2850; eta: 30; loss: multiclass_nll; loss_params: {alpha: 1, margin: 1}; optimizer: adam; lr: 0.0001; regularizer: LP; regularizer_params: {'p': 3, 'lambda': 0.0}; seed: 0; batches_count: 110

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	IMR	MRR	Hits@1	Hits@3	Blits@1	Hyperparameters
TransE45	227	0.62	0.48	0.72	0.84	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; seed: 0; batches_count: 100;
DistMult	227	0.71	0.66	0.75	0.80	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;
ComplEx	199	0.73	0.68	0.77	0.82	k: 200; epochs: 4000; eta: 20; loss: self_adversarial; loss_params: margin: 1; optimizer: adam; optimizer_params: lr: 0.0005; regularizer: LP; regularizer_params: lambda: 0.0001; p: 3; seed: 0; batches_count: 100;
HoleE	238	0.73	0.67	0.77	0.82	k: 200; epochs: 4000; eta: 20; loss: self_adversarial; loss_params: margin: 1; optimizer: adam; optimizer_params: lr: 0.0005; seed: 0; batches_count: 20;
RotatE	222	0.70	0.59	0.80	0.88	k: 200; batch_size: 97, epochs: 1425, eta: 20, loss: self_adversarial, loss_params: {'margin': 5, 'alpha': 1.0}, regularizer_params: {'p': 3, 'lambda': 0.001}, optimizer: adam, learning_rate: 1e-05

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	IMR	MRR	Hits@1	Hits@3	Blits@1	Hyperparameters
TransE278	699	0.66	0.42	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; seed: 0; batches_count: 100;
DistMult	699	0.82	0.71	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; batches_count: 50;
ComplEx	713	0.94	0.93	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: 20;
HoleE	676	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;
RotatE	222	0.95	0.94	0.96	0.97	k: 200; epochs: 1425; k: 200; eta: 20; loss: self_adversarial loss_params: {'margin': 5, 'alpha': 1.0}; optimizer: 'adam'; learning_rate: 1e-05; regularizer: LP; regularizer_params: {'p': 3, 'lambda': 0.001}; batches_count: 29

To reproduce the above results:

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

Note: Running `predictive_performance.py` on all datasets, for all models takes ~34 hours on an Intel Xeon Gold 6226R, 256 GB, equipped with Tesla A100 40GB GPUs and Ubuntu 20.04.

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 can save a lot of training time, but it also adds some computational cost 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.

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,rotate}]

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,rotate}, --model {complex,transe,distmult,hole,rotate}
```

3.7.7 Loading Pre-Trained Models

If you want to load the pre-trained models used to obtain the above performance, have a look at `load_pretrained_model()`.

3.7.8 Runtime Performance

Training the models on FB15K-237 (`k=100`, `eta=10`, `batches_count=10`, `loss=multiclass_nll`), on an Intel Xeon Gold 6226R, 256 GB, equipped with Tesla A100 40GB GPUs and Ubuntu 20.04 gives the following runtime report:

model	seconds/epoch
ComplEx	0.18
RotatE	0.19
TransE	0.09
DistMult	0.10
HolE	0.18

3.8 Bibliography

3.9 Changelog

3.9.1 2.1.0

28 February 2024

- Addition of RotatE to the available scoring functions.
- Addition of a module to load models (TransE, DistMult, ComplEx, RotatE, HolE) pre-trained on benchmark datasets (fb15k-237, wn18rr, yago3-10, fb15k, wn18).
- Improved efficiency of the validation.
- Other minor efficiency improvements, fixes and code clean-up.

3.9.2 2.0.1

12 July 2023

- Fixed bug preventing the saving of calibrated models.
- Extended type support for the predict method to list of triples.
- Updated experiments performance.
- Minor fixes.

3.9.3 2.0.0

7 March 2023

- Switched to TensorFlow 2 back-end.
- Keras style APIs.
- Unique model class ScoringBasedEmbeddingModel for all scoring functions that can be specified as a parameter when initializing the class.
- Change of the data input/output pipeline.
- Extension of supported optimizers, regularizers and initializer.
- Different data storage support: no-backend (in memory) and SQLite-based backend.
- Codex-M Knowledge Graph included in the APIs for automatic download.
- ConvKB, ConvE, ConvE(1-N) not supported anymore as they are computationally expensive and thus not commonly used.
- Support AmpliGraph 1.4 API within `ampligraph.compat` module.

3.9.4 1.4.0

26 May 2021

- Added support for numerical attributes on edges (FocusE) (#235)
- Added loaders for benchmark datasets with numeric values on edges (O*NET20K, PPI5K, NL27K, CN15K)
- Added discovery API to find nearest neighbors in embedding space (#240)
- Change of optimizer (from BFSG to Adam) to calibrate models with ground truth negatives (#239)
- 10x speed improvement on train_test_split_unseen API (#242)
- Added support to visualize training progression via tensorboard (#230)
- Bug fix in large graph mode (when evaluate_performance with entities_subset is used) (#231)
- Updated save model api to save embedding matrix > 6GB (#233)
- Doc updates (#247, #221)
- Fixed ntriples loader spurious trailing dot.
- Add tensorboard_logs_path to model.fit() for tracking training loss and early stopping criteria.

3.9.5 1.3.2

25 Aug 2020

- Added constant initializer (#205)
- Ranking strategies for breaking ties (#212)
- ConvE Bug Fixes (#210, #194)
- Efficient batch sampling (#202)
- Added pointer to documentation for large graph mode and Docs for Optimizer (#216)

3.9.6 1.3.1

18 Mar 2020

- Minor bug fix in ConvE (#189)

3.9.7 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

3.9.8 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)

3.9.9 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

3.9.10 1.0.3

7 Jun 2019

- Fixed regression in RandomBaseline (#94)
- Added TensorBoard Embedding Projector support (#86)
- Minor bugfixing (#87, #47)

3.9.11 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)

3.9.12 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

3.9.13 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 developed and 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
          Alberto Bernardi and
          Adrianna Janik and
          Aldan Creo 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)

- Alberto Bernardi
- Luca Costabello
- Aldan Creo
- Adrianna Janik

Past contributors

- Nicholas McCarthy
- Rory McGrath
- Chan Le Van
- Sumit Pai
- Pedro Tabacof

3.10.4 License

AmpliGraph is licensed under the Apache 2.0 License.

BIBLIOGRAPHY

- [ABK+07] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. Dbpedia: a nucleus for a web of open data. In *The semantic web*, 722–735. Springer, 2007.
- [BHBL11] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked data: the story so far. In *Semantic services, interoperability and web applications: emerging concepts*, 205–227. IGI Global, 2011.
- [BUGD+13] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. Translating embeddings for modeling multi-relational data. In *Advances in neural information processing systems*, 2787–2795. 2013.
- [DMSR18] Tim Dettmers, Pasquale Minervini, Pontus Stenetorp, and Sebastian Riedel. Convolutional 2d knowledge graph embeddings. In *Procs of AAAI*. 2018. URL: <https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/17366>.
- [HOSM17] Takuo Hamaguchi, Hidekazu Oiwa, Masashi Shimbo, and Yuji Matsumoto. Knowledge transfer for out-of-knowledge-base entities: A graph neural network approach. *IJCAI International Joint Conference on Artificial Intelligence*, pages 1802–1808, 2017.
- [HS17] Katsuhiko Hayashi and Masashi Shimbo. On the equivalence of holographic and complex embeddings for link prediction. *CoRR*, 2017. URL: <http://arxiv.org/abs/1702.05563>, arXiv:1702.05563.
- [NRP+16] Maximilian Nickel, Lorenzo Rosasco, Tomaso A Poggio, and others. Holographic embeddings of knowledge graphs. In *AAAI*, 1955–1961. 2016.
- [PC21] Sumit Pai and Luca Costabello. Learning Embeddings from Knowledge Graphs With Numeric Edge Attributes. In *IJCAI*. 2021. URL: <https://arxiv.org/abs/2105.08683>.
- [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. URL: <https://arxiv.org/abs/1912.10000>.
- [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

C

ampligraph.compat, 32

D

ampligraph.datasets, 10
ampligraph.discovery, 34

E

ampligraph.evaluation, 33

I

ampligraph.latent_features, 28

P

ampligraph.pretrained_models, 50

U

ampligraph.utils, 45

INDEX

A

ampligraph.compat
 module, 32
ampligraph.datasets
 module, 10
ampligraph.discovery
 module, 34
ampligraph.evaluation
 module, 33
ampligraph.latent_features
 module, 28
ampligraph.pretrained_models
 module, 50
ampligraph.utils
 module, 45

C

create_tensorboard_visualizations() (in module
 ampligraph.utils), 46

D

dataframe_to_triples() (in module ampli-
 graph.utils), 49
discover_facts() (in module ampligraph.discovery),
 34

F

find_clusters() (in module ampligraph.discovery), 37
find_duplicates() (in module ampligraph.discovery),
 40

L

load_cn15k() (in module ampligraph.datasets), 27
load_codex() (in module ampligraph.datasets), 19
load_fb13() (in module ampligraph.datasets), 18
load_fb15k() (in module ampligraph.datasets), 20
load_fb15k_237() (in module ampligraph.datasets), 13
load_from_csv() (in module ampligraph.datasets), 11
load_from_ntriples() (in module ampli-
 graph.datasets), 12
load_from_rdf() (in module ampligraph.datasets), 12
load_nl27k() (in module ampligraph.datasets), 26

load_onet20k() (in module ampligraph.datasets), 23
load_ppi5k() (in module ampligraph.datasets), 24
load_wn11() (in module ampligraph.datasets), 17
load_wn18() (in module ampligraph.datasets), 21
load_wn18rr() (in module ampligraph.datasets), 15
load_yago3_10() (in module ampligraph.datasets), 16

M

module
 ampligraph.compat, 32
 ampligraph.datasets, 10
 ampligraph.discovery, 34
 ampligraph.evaluation, 33
 ampligraph.latent_features, 28
 ampligraph.pretrained_models, 50
 ampligraph.utils, 45

P

preprocess_focusE_weights() (in module ampli-
 graph.utils), 49

Q

query_topn() (in module ampligraph.discovery), 44

R

restore_model() (in module ampligraph.utils), 46

S

save_model() (in module ampligraph.utils), 45