Getting Started with AmpliGraph

In this tutorial we will demonstrate how to use the AmpliGraph library.

Things we will cover:

1. Exploration of a graph dataset
2. Splitting graph datasets into train and test sets
3. Training a model
4. Model selection and hyper-parameter search
5. Saving and restoring a model
6. Evaluating a model
7. Using link prediction to discover unknown relations
8. Visualizing embeddings using Tensorboard

Requirements

A Python environment with the AmpliGraph library installed. Please follow the install guide.

Some sanity check:

In [7]:
```python
import numpy as np
import pandas as pd
import ampligraph

ampligraph.__version__
```
Out[7]:
```
'1.0.3'
```

1. Dataset exploration

First things first! Lets import the required libraries and retrieve some data.

In this tutorial we're going to use the **Game of Thrones knowledge Graph**. Please note: this isn't the *greatest* dataset for demonstrating the power of knowledge graph embeddings, but is small, intuitive and should be familiar to most users.

We downloaded the neo4j graph published here. Such dataset has been generated using these APIs which expose in a machine-readable fashion the content of open free sources such as A Wiki of Ice and Fire. We discarded all properties and saved all the directed, labeled relations in a plaintext file. Each relation (i.e. a triple) is in the form:

```
<subject, predicate, object>
```

The schema of the graph looks like this (image from neo4j-examples/game-of-thrones):

Run the following cell to pull down the dataset and load it in memory with AmpliGraph `load_from_csv()` utility function:

In [8]:
```python
import requests
from ampligraph.datasets import load_from_csv

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=',')
X[:5, ]
```
Out[8]:
```
array([['Smithyton', 'SEAT_OF', 'House Shermer of Smithyton'],
       ['House Mormont of Bear Island', 'LED_BY', 'Maege Mormont'],
       ['Margaery Tyrell', 'SPOUSE', 'Joffrey Baratheon'],
       ['Maron Nymeros Martell', 'ALLIED_WITH',
        'House Nymeros Martell of Sunspear'],
       ['House Gargalen of Salt Shore', 'IN_REGION', 'Dorne']],
      dtype=object)
```

Let's list the subject and object entities found in the dataset:

In [9]:
```python
entities = np.unique(np.concatenate([X[:, 0], X[:, 2]]))
entities
```
Out[9]:
```
array(['Abelar Hightower', 'Acorn Hall', 'Addam Frey', ..., 'the Antlers',
       'the Paps', 'unnamed tower'], dtype=object)
```

.. and all of the relationships that link them. Remember, these relationships only link *some* of the entities.

In [10]:
```python
relations = np.unique(X[:, 1])
relations
```
Out[10]:
```
array(['ALLIED_WITH', 'BRANCH_OF', 'FOUNDED_BY', 'HEIR_TO', 'IN_REGION',
       'LED_BY', 'PARENT_OF', 'SEAT_OF', 'SPOUSE', 'SWORN_TO'],
      dtype=object)
```

2. Defining train and test datasets

As is typical in machine learning, we need to split our dataset into training and test (and sometimes validation) datasets.

What differs from the standard method of randomly sampling N points to make up our test set, is that our data points are two entities linked by some relationship, and we need to take care to ensure that all entities are represented in train and test sets by at least one triple.

To accomplish this, AmpliGraph provides the `train_test_split_no_unseen` function.

For sake of example, we will create a small test size that includes only 100 triples:

In [11]:
```python
from ampligraph.evaluation import train_test_split_no_unseen

X_train, X_test = train_test_split_no_unseen(X, test_size=100)
```

Our data is now split into train/test sets. If we need to further divide into a validation dataset we can just repeat using the same procedure on the test set (and adjusting the split percentages).

In [12]:
```python
print('Train set size: ', X_train.shape)
print('Test set size: ', X_test.shape)
```
```
Train set size:  (3075, 3)
Test set size:  (100, 3)
```

3. Training a model

AmpliGraph has implemented several Knoweldge Graph Embedding models (TransE, ComplEx, DistMult, HolE), but to begin with we're just going to use the ComplEx model (with default values), so lets import that:

In [13]:
```python
from ampligraph.latent_features import ComplEx
```

Lets go through the parameters to understand what's going on:

- `k` : the dimensionality of the embedding space
- `eta` (η) : the number of negative, or false triples that must be generated at training runtime for each positive, or true triple
- `batches_count` : the number of batches in which the training set is split during the training loop. If you are having into low memory issues than settings this to a higher number may help.
- `epochs` : the number of epochs to train the model for.
- `optimizer` : the Adam optimizer, with a learning rate of 1e-3 set via the *optimizer_params* kwarg.
- `loss` : pairwise loss, with a margin of 0.5 set via the *loss_params* kwarg.
- `regularizer` : L_p regularization with $p = 2$, i.e. l2 regularization. λ = 1e-5, set via the *regularizer_params* kwarg.

Now we can instantiate the model:

In [14]:
```python
model = ComplEx(batches_count=100,
                seed=0,
                epochs=200,
                k=150,
                eta=5,
                optimizer='adam',
                optimizer_params={'lr':1e-3},
                loss='multiclass_nll',
                regularizer='LP',
                regularizer_params={'p':3, 'lambda':1e-5},
                verbose=True)
```

Filtering negatives

AmpliGraph aims to follow scikit-learn's ease-of-use design philosophy and simplify everything down to `fit`, `evaluate`, and `predict` functions.

However, there are some knowledge graph specific steps we must take to ensure our model can be trained and evaluated correctly. The first of these is defining the filter that will be used to ensure that no *negative* statements generated by the corruption procedure are actually positives. This is simply done by concatenating our train and test sets. Now when negative triples are generated by the corruption strategy, we can check that they aren't actually true statements.

In [15]:
```python
positives_filter = X
```

Fitting the model

Once you run the next cell the model will train.

On a modern laptop this should take ~3 minutes (although your mileage may vary, especially if you've changed any of the hyper-parameters above).

In [16]:
```python
import tensorflow as tf
tf.logging.set_verbosity(tf.logging.ERROR)

model.fit(X_train, early_stopping = False)
```
```
Average Loss:   0.033250: 100%|██████████| 200/200 [02:54<00:00,  1.07epoch/s]
```

5. Saving and restoring a model

Before we go any further, let's save the best model found so that we can restore it in future.

In [22]:
```python
from ampligraph.latent_features import save_model, restore_model
```
In [23]:
```python
save_model(model, './best_model.pkl')
```

This will save the model in the ampligraph_tutorial directory as `best_model.pkl`.

.. we can then delete the model ..

In [24]:
```python
del model
```

.. and then restore it from disk! Ta-da!

In [25]:
```python
model = restore_model('./best_model.pkl')
```

And let's just double check that the model we restored has been fit:

In [26]:
```python
if model.is_fitted:
    print('The model is fit!')
else:
    print('The model is not fit! Did you skip a step?')
```
```
The model is fit!
```

6. Evaluating a model

Now it's time to evaluate our model on the test set to see how well it's performing.

For this we'll use the `evaluate_performance` function:

In [27]:
```python
from ampligraph.evaluation import evaluate_performance
```

And let's look at the arguments to this function:

- `X` - the data to evaluate on. We're going to use our test set to evaluate.
- `model` - the model we previously trained.
- `filter_triples` - will filter out the false negatives generated by the corruption strategy.
- `use_default_protocol` - specifies whether to use the default corruption protocol. If True, then subj and obj are corrupted separately during evaluation.
- `verbose` - will give some nice log statements. Let's leave it on for now.

Running evaluation

In [37]:
```python
ranks = evaluate_performance(X_test,
                             model=model,
                             filter_triples=positives_filter,   # Corruption strategy filter de
fined above
                             use_default_protocol=True, # corrupt subj and obj separately while
evaluating
                             verbose=True)
```
```
100%|██████████| 100/100 [00:01<00:00, 57.37it/s]
```

The ***ranks*** returned by the evaluate_performance function indicate the rank at which the test set triple was found when performing link prediction using the model.

For example, given the triple:

```
<House Stark of Winterfell, IN_REGION The North>
```

The model returns a rank of 7. This tells us that while it's not the highest likelihood true statement (which would be given a rank 1), it's pretty likely.

Metrics

Let's compute some evaluate metrics and print them out.

We're going to use the mrr_score (mean reciprocal rank) and hits_at_n_score functions.

- ***mrr_score***: The function computes the mean of the reciprocal of elements of a vector of rankings ranks.
- ***hits_at_n_score***: The function computes how many elements of a vector of rankings ranks make it to the top n positions.

In [38]:
```python
from ampligraph.evaluation import mr_score, mrr_score, hits_at_n_score

mrr = mrr_score(ranks)
print("MRR: %.2f" % (mrr))

hits_10 = hits_at_n_score(ranks, n=10)
print("Hits@10: %.2f" % (hits_10))
hits_3 = hits_at_n_score(ranks, n=3)
print("Hits@3: %.2f" % (hits_3))
hits_1 = hits_at_n_score(ranks, n=1)
print("Hits@1: %.2f" % (hits_1))
```
```
MRR: 0.46
Hits@10: 0.58
Hits@3: 0.53
Hits@1: 0.38
```

Now, how do we interpret those numbers?

Hits@N indicates how many times in average a true triple was ranked in the top-N. Therefore, on average, we guessed the correct subject or object 53% of the time when considering the top-3 better ranked triples. The choice of which N makes more sense depends on the application.

The Mean Reciprocal Rank (MRR) is another popular metrics to assess the predictive power of a model.

7. Predicting New Links

Link prediction allows us to infer missing links in a graph. This has many real-world use cases, such as predicting connections between people in a social network, interactions between proteins in a biological network, and music recommendation based on prior user taste.

In our case, we're going to see which of the following candidate statements (that we made up) are more likely to be true:

In [22]:
```python
X_unseen = np.array([
    ['Jorah Mormont', 'SPOUSE', 'Daenerys Targaryen'],
    ['Tyrion Lannister', 'SPOUSE', 'Missandei'],
    ["King's Landing", 'SEAT_OF', 'House Lannister of Casterly Rock'],
    ['Sansa Stark', 'SPOUSE', 'Petyr Baelish'],
    ['Daenerys Targaryen', 'SPOUSE', 'Jon Snow'],
    ['Daenerys Targaryen', 'SPOUSE', 'Craster'],
    ['House Stark of Winterfell', 'IN_REGION', 'The North'],
    ['House Stark of Winterfell', 'IN_REGION', 'Dorne'],
    ['House Tyrell of Highgarden', 'IN_REGION', 'Beyond the Wall'],
    ['Brandon Stark', 'ALLIED_WITH', 'House Stark of Winterfell'],
    ['Brandon Stark', 'ALLIED_WITH', 'House Lannister of Casterly Rock'],
    ['Rhaegar Targaryen', 'PARENT_OF', 'Jon Snow'],
    ['House Hutcheson', 'SWORN_TO', 'House Tyrell of Highgarden'],
    ['Daenerys Targaryen', 'ALLIED_WITH', 'House Stark of Winterfell'],
    ['Daenerys Targaryen', 'ALLIED_WITH', 'House Lannister of Casterly Rock'],
    ['Jaime Lannister', 'PARENT_OF', 'Myrcella Baratheon'],
    ['Robert I Baratheon', 'PARENT_OF', 'Myrcella Baratheon'],
    ['Cersei Lannister', 'PARENT_OF', 'Myrcella Baratheon'],
    ['Cersei Lannister', 'PARENT_OF', 'Brandon Stark'],
    ["Tywin Lannister", 'PARENT_OF', 'Jaime Lannister'],
    ["Missandei", 'SPOUSE', 'Grey Worm'],
    ["Brienne of Tarth", 'SPOUSE', 'Jaime Lannister']
])
```
In [23]:
```python
unseen_filter = np.array(list({tuple(i) for i in np.vstack((positives_filter, X_unseen))}))
```
In [24]:
```python
ranks_unseen = evaluate_performance(
    X_unseen,
    model=model,
    filter_triples=unseen_filter,   # Corruption strategy filter defined above
    corrupt_side = 's+o',
    use_default_protocol=False, # corrupt subj and obj separately while evaluating
    verbose=True
)
```
```
100%|██████████| 22/22 [00:00<00:00, 183.80it/s]
```
In [25]:
```python
scores = model.predict(X_unseen)
```
```
100%|██████████| 22/22 [00:00<00:00, 365.01it/s]
```

We transform the scores (real numbers) into probabilities (bound between 0 and 1) using the expit transform.

Note that the probabilities are not calibrated in any sense.

Advanced note: To calibrate the probabilities, one may use a procedure such as Platt scaling or Isotonic regression. The challenge is to define what is a true triple and what is a false one, as the calibration of the probability of a triple being true depends on the base rate of positives and negatives.

In [26]:
```python
from scipy.special import expit
probs = expit(scores)
```
In [27]:
```python
pd.DataFrame(list(zip([' '.join(x) for x in X_unseen],
                      ranks_unseen,
                      np.squeeze(scores),
                      np.squeeze(probs))),
             columns=['statement', 'rank', 'score', 'prob']).sort_values("score")
```
Out[27]:

	statement	rank	score	prob
5	Daenerys Targaryen SPOUSE Craster	4090	-2.750880	0.060037
10	Brandon Stark ALLIED_WITH House Lannister of C...	3821	-1.515378	0.180143
18	Cersei Lannister PARENT_OF Brandon Stark	4061	-1.386417	0.199980
1	Tyrion Lannister SPOUSE Missandei	3190	-0.554477	0.364826
8	House Tyrell of Highgarden IN_REGION Beyond th...	3075	-0.452406	0.388789
15	Jaime Lannister PARENT_OF Myrcella Baratheon	3257	-0.408462	0.399281
21	Brienne of Tarth SPOUSE Jaime Lannister	2860	-0.384484	0.405046
11	Rhaegar Targaryen PARENT_OF Jon Snow	2942	-0.199926	0.450184
4	Daenerys Targaryen SPOUSE Jon Snow	2042	-0.035172	0.491208
13	Daenerys Targaryen ALLIED_WITH House Stark of ...	1010	0.012399	0.503100
9	Brandon Stark ALLIED_WITH House Stark of Winte...	1302	0.038987	0.509745
0	Jorah Mormont SPOUSE Daenerys Targaryen	1523	0.124081	0.530981
14	Daenerys Targaryen ALLIED_WITH House Lannister...	694	0.143645	0.535850
17	Cersei Lannister PARENT_OF Myrcella Baratheon	889	0.402498	0.599288
2	King's Landing SEAT_OF House Lannister of Cast...	400	0.861543	0.702983
7	House Stark of Winterfell IN_REGION Dorne	136	1.118141	0.753644
19	Tywin Lannister PARENT_OF Jaime Lannister	87	1.220659	0.772180
16	Robert I Baratheon PARENT_OF Myrcella Baratheon	53	1.697480	0.845205
6	House Stark of Winterfell IN_REGION The North	7	2.445815	0.920255
20	Missandei SPOUSE Grey Worm	73	2.593231	0.930425
3	Sansa Stark SPOUSE Petyr Baelish	29	3.190537	0.960477
12	House Hutcheson SWORN_TO House Tyrell of Highg...	3	4.680758	0.990813

We see that the embeddings captured some truths about Westeros. For example, **House Stark is placed in the North rather than Dorne**. It also realises **Daenerys Targaryen has no relation with Craster, nor Tyrion with Missandei**. It captures random trivia, as **House Hutcheson is indeed in the Reach and sworn to the Tyrells**. On the other hand, some marriages that it predicts never really happened. These mistakes are understandable: those characters were indeed close and appeared together in many different circumstances.

8. Visualizing Embeddings with Tensorboard projector

The kind folks at Google have created Tensorboard, which allows us to graph how our model is learning (or .. not :|), peer into the innards of neural networks, and visualize high-dimensional embeddings in the browser.

Lets import the `create_tensorboard_visualization` function, which simplifies the creation of the files necessary for Tensorboard to display the embeddings.

In [39]:
```python
from ampligraph.utils import create_tensorboard_visualizations
```

And now we'll run the function with our model, specifying the output path:

In [40]:
```python
create_tensorboard_visualizations(model, 'GoT_embeddings')
```

If all went well, we should now have a number of files in the `AmpliGraph/tutorials/GoT_embeddings` directory:

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

To visualize the embeddings in Tensorboard, run the following from your command line inside `AmpliGraph/tutorials`:

```
tensorboard --logdir=./visualizations
```

.. and once your browser opens up you should be able to see and explore your embeddings as below (PCA-reduced, two components):

The End

You made it to the end! Well done!

For more information please visit the AmpliGraph GitHub (and remember to star the project!), or check out the documentation