Using `AmpliGraph` to generate GoT Knowledge Graph Embeddings

Content of this notebook was prepared by Basel Shbita (shbita@usc.edu) as part of the class DSCI 558: Building Knowledge Graphs during Fall 2020 at University of Southern California (USC).

Notes:

- You are supposed to write your code or modify our code in any cell starting with `# ** STUDENT CODE`.
- Much content of this notebook was borrowed from AmpliGraph tutorials

`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. It can be used to generate stand-alone knowledge graph embeddings, discover new knowledge from an existing knowledge graph and complete large knowledge graphs with missing statements.

In this task, you will gain some hands-on experience working with Knowledge Graph Embeddings. Specifically, you will use the *TransE*, *DistMult* and *ComplEx* models to learn the embeddings of a (small) KG. You will be required to split the dataset to train and test sets, train the model, evaluate it and then generate a visualization for each model type!

In [1]:
```python
import numpy as np
import pandas as pd
import ampligraph

ampligraph.__version__
```

Out[1]: '1.3.2'

Importing the dataset

We will use the Game of Thrones (reduced) Knowledge Graph found in file `GoT.csv`.
Each relation (i.e. a triple) is in the form: `<subject, predicate, object>`

Run the following cell to load the dataset in memory with using the `load_from_csv()` utility function:

In [3]:
```python
from ampligraph.datasets import load_from_csv

X = load_from_csv('.', 'GoT.csv', sep=',') # numpy.ndarray; size (3175,3)
```

In [5]:
```python
# inspect the top triples:
pd.DataFrame(X, columns=['s', 'p', 'o']).head()
```

Out[5]:

	s	p	o
0	Smithyton	SEAT_OF	House Shermer of Smithyton
1	House Mormont of Bear Island	LED_BY	Maege Mormont
2	Margaery Tyrell	SPOUSE	Joffrey Baratheon
3	Maron Nymeros Martell	ALLIED_WITH	House Nymeros Martell of Sunspear
4	House Gargalen of Salt Shore	IN_REGION	Dorne

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

In [6]:
```python
# an array of unique subjects and objects => size (2050,)
entities = np.unique(np.concatenate([X[:, 0], X[:, 2]]))
entities
```

Out[6]:
```
array(['Abelar Hightower', 'Acorn Hall', 'Addam Frey', ..., 'the Antlers',
       'the Paps', 'unnamed tower'], dtype=object)
```

.. and all of the relationships that link them.

In [7]:
```python
 # an array of unique preicates => size (10,)
relations = np.unique(X[:, 1])
relations
```

Out[7]:
```
array(['ALLIED_WITH', 'BRANCH_OF', 'FOUNDED_BY', 'HEIR_TO', 'IN_REGION',
       'LED_BY', 'PARENT_OF', 'SEAT_OF', 'SPOUSE', 'SWORN_TO'],
      dtype=object)
```

Defining train and test datasets

As is typical in machine learning, we need to split our dataset into training and test sets.

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.

In [8]:
```python
from ampligraph.evaluation import train_test_split_no_unseen

# we create a 10% test set split
X_train, X_test = train_test_split_no_unseen(X, test_size=int(X.shape[0]/10))
```

Our data is now split into train/test sets:

In [9]:
```python
print('Train set size: ', X_train.shape)
print('Test set size: ', X_test.shape)
```

```
Train set size:  (2858, 3)
Test set size:  (317, 3)
```

Task 2.1

Task 2.1.x.1 Training the model

`AmpliGraph` has implemented several Knoweldge Graph Embedding models (*TransE, ComplEx, DistMult, etc...*):

In [10]:
```python
from ampligraph.latent_features import TransE, DistMult, 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 [11]:
```python
# ** STUDENT CODE
# TODO: try different model types: TransE [2.1.1], DistMult [2.1.2], ComplEx [2.1.3]
EmbeddingMethod = ComplEx
```

In [12]:
```python
model = EmbeddingMethod(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 [13]:
```python
positives_filter = X
```

Fitting the model

Once you run the next cell the model will train:

In [14]:
```python
import tensorflow as tf
tf.logging.set_verbosity(tf.logging.ERROR)

model.fit(X_train, early_stopping = False)
```

```
Average Loss:   0.016231: 100%|██████████| 200/200 [05:55<00:00,  1.78s/epoch]
```

2.1.x.2 Evaluating the 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 [15]:
```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.

Let's run some evaluations:

In [16]:
```python
ranks = evaluate_performance(X_test,
                             model=model,
                             filter_triples=positives_filter,
                             use_default_protocol=True,
                             verbose=True)
```

```
WARNING - DeprecationWarning: use_default_protocol will be removed in future. Please use corrupt_side argument ins
tead.
100%|██████████| 317/317 [00:06<00:00, 50.56it/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, if we run the triple `<House Stark of Winterfell, IN_REGION, The North>` and the model returns a rank of `7`, it tells us that while it's not the highest likelihood true statement (which would be given a rank 1), it's pretty likely.

metrics

For the evaluation metrics, we are 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 [17]:
```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: 0.35
Hits@10: 0.47
Hits@3: 0.38
Hits@1: 0.29
```

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.

^ Please note that a screenshot of these scores are required for task of 2.1.x.2 ^

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 are going to see which of the following candidate statements are more likely to be true:

In [18]:
```python
X_unseen = np.array([
    ['Jorah Mormont', 'SPOUSE', 'Daenerys Targaryen'],
    ["King's Landing", 'SEAT_OF', 'House Lannister of Casterly Rock'],
    ['Brienne of Tarth', 'SPOUSE', 'Jaime Lannister'],
    ['House Stark of Winterfell', 'IN_REGION', 'The North'],
])
```

In [19]:
```python
unseen_filter = np.array(list({tuple(i) for i in np.vstack((positives_filter, X_unseen))}))
```

In [20]:
```python
ranks_unseen = evaluate_performance(
    X_unseen,
    model=model,
    filter_triples=unseen_filter,
    corrupt_side = 's+o',
    use_default_protocol=False,
    verbose=True
)
```

```
100%|██████████| 4/4 [00:00<00:00, 25.36it/s]
```

In [21]:
```python
scores = model.predict(X_unseen)
```

We transform the scores (real numbers) into probabilities (bound between 0 and 1) using the `expit` transform (note that the probabilities are not calibrated).

> 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 [22]:
```python
from scipy.special import expit
probs = expit(scores)
```

In [23]:
```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", ascending=False)
```

Out[23]:

	statement	rank	score	prob
3	House Stark of Winterfell IN_REGION The North	177	1.308594	0.787278
1	King's Landing SEAT_OF House Lannister of Cast...	1331	0.196228	0.548900
0	Jorah Mormont SPOUSE Daenerys Targaryen	2232	0.039165	0.509790
2	Brienne of Tarth SPOUSE Jaime Lannister	2430	-0.124812	0.468837

Task 2.1.x.3: Visualizing Embeddings with Tensorboard projector

we can now visualize the 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 [24]:
```python
from ampligraph.utils import create_tensorboard_visualizations
```

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

In [25]:
```python
create_tensorboard_visualizations(model, 'dsci558_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:

```
tensorboard  --logdir="./dsci558_embeddings"
```

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

^ Please note that a screenshot of embedding visualization is required for task 2.1.x.3 ^