Tutorial_A_DistMult

January 15, 2024

1 Machine Learning for Graphs - Tutorial A: Triple Scoring with DistMult

Fill in your names and group number here:

NAME STUDENT A : Alexandru Dochian (2776000)

NAME STUDENT B : Matteo De Rizzo (2749303)

GROUP NUMBER : 6

Implementing a machine learning experiment with graph data is an important skill that you will learn as part of this course. This hands-on tutorial will help you develop this skill, as well as help you familiarize yourself with many of the steps and techniques that you will likely need to use for your final project.

The oldest and arguably most well-studied machine learning task for graphs is that of *link prediction*. The goal of this task is, as the name implies, to predict the existence of edges in a graph. Since real-world graphs often have large gaps in the information they encode, represented by missing links, and because this task has the potential of predicting these missing links, link prediction is also known as *knowledge base completion*.

For this tutorial, you are asked to implement a link prediction model. Once correctly implemented, you are to train and test the model on a graph-based dataset that you have prepared during the first half of the tutorial. To help you on your way, we have already prepared this Python Notebook.

You are asked to team up with another student and to work together on this tutorial. Please register your team by creating a new group and by adding both members.

1.1 Working with Graphs

Most major programming languages do not support graph data out of the box. Therefore, before we can continue, we first need to download and install the RDFlib package, which extends Python with an interface to graphs that are encoded using the *Resource Description Format* (RDF). The RDF data model is a W3C open standard to encode knowledge, information, and data in graph form, and is a common preferred choice for creating *knowledge graphs*. Rather than using the common prefix notation for links, i.e., `relation(head, tail)`, RDF knowledge graphs use the *in-fix* notation: `(head, relation, tail)`. Because of this, the links in a knowledge graph are often also called *triples*.

Run the cells below to install the RDFlib package in your Python environment and to import and inspect the graph

[ ]:
```
# install the rdflib package
%pip install rdflib
```

```
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: rdflib in
/home/samanu/.local/lib/python3.10/site-packages (7.0.0)
Requirement already satisfied: pyparsing<4,>=2.1.0 in /usr/lib/python3/dist-
packages (from rdflib) (2.4.7)
Requirement already satisfied: isodate<0.7.0,>=0.6.0 in
/home/samanu/.local/lib/python3.10/site-packages (from rdflib) (0.6.1)
Requirement already satisfied: six in /usr/lib/python3/dist-packages (from
isodate<0.7.0,>=0.6.0->rdflib) (1.16.0)
Note: you may need to restart the kernel to use updated packages.
```

[ ]:
```
import gzip
import rdflib
```

[ ]:
```
# read the data from disk

g_train = rdflib.Graph()
g_test = rdflib.Graph()

path = './data/'
with gzip.open(path + f'wn18rr_train.nt.gz', 'r') as gf:
    g_train.parse(data = gf.read(), format = 'nt')
with gzip.open(path + f'wn18rr_test.nt.gz', 'r') as gf:
    g_test.parse(data = gf.read(), format = 'nt')
```

[ ]:
```
# test whether reading the data was successful by printing 10 links from the␣
 ↪train and test set each

print('TRAIN:', end='\n')
for h,r,t in list(g_train)[:10]:
    print(f'{h} {r} {t}', end='\n')

print('TEST:', end='\n')
for h,r,t in list(g_test)[:10]:
    print(f'{h} {r} {t}', end='\n')
```

```
TRAIN:
http://wordnet-rdf.princeton.edu/id/00785690 http://wordnet-
rdf.princeton.edu/ontology#_derivationally_related_form http://wordnet-
rdf.princeton.edu/id/10611613
http://wordnet-rdf.princeton.edu/id/05501185 http://wordnet-
rdf.princeton.edu/ontology#_hypernym http://wordnet-
```

rdf.princeton.edu/id/05462674
http://wordnet-rdf.princeton.edu/id/01824575 http://wordnet-
rdf.princeton.edu/ontology#_hypernym http://wordnet-
rdf.princeton.edu/id/01823013
http://wordnet-rdf.princeton.edu/id/09973209 http://wordnet-
rdf.princeton.edu/ontology#_hypernym http://wordnet-
rdf.princeton.edu/id/09972661
http://wordnet-rdf.princeton.edu/id/14474052 http://wordnet-
rdf.princeton.edu/ontology#_derivationally_related_form http://wordnet-
rdf.princeton.edu/id/02331262
http://wordnet-rdf.princeton.edu/id/01811682 http://wordnet-
rdf.princeton.edu/ontology#_member_meronym http://wordnet-
rdf.princeton.edu/id/01812471
http://wordnet-rdf.princeton.edu/id/03027250 http://wordnet-
rdf.princeton.edu/ontology#_has_part http://wordnet-
rdf.princeton.edu/id/03594277
http://wordnet-rdf.princeton.edu/id/09810867 http://wordnet-
rdf.princeton.edu/ontology#_hypernym http://wordnet-
rdf.princeton.edu/id/09979589
http://wordnet-rdf.princeton.edu/id/01271189 http://wordnet-
rdf.princeton.edu/ontology#_hypernym http://wordnet-
rdf.princeton.edu/id/01332730
http://wordnet-rdf.princeton.edu/id/11700401 http://wordnet-
rdf.princeton.edu/ontology#_member_meronym http://wordnet-
rdf.princeton.edu/id/11700676
TEST:
http://wordnet-rdf.princeton.edu/id/01473990 http://wordnet-
rdf.princeton.edu/ontology#_hypernym http://wordnet-
rdf.princeton.edu/id/08103777
http://wordnet-rdf.princeton.edu/id/07763290 http://wordnet-
rdf.princeton.edu/ontology#_hypernym http://wordnet-
rdf.princeton.edu/id/07705931
http://wordnet-rdf.princeton.edu/id/13627516 http://wordnet-
rdf.princeton.edu/ontology#_has_part http://wordnet-
rdf.princeton.edu/id/13627327
http://wordnet-rdf.princeton.edu/id/12356395 http://wordnet-
rdf.princeton.edu/ontology#_hypernym http://wordnet-
rdf.princeton.edu/id/12205694
http://wordnet-rdf.princeton.edu/id/01346003 http://wordnet-
rdf.princeton.edu/ontology#_derivationally_related_form http://wordnet-
rdf.princeton.edu/id/03848348
http://wordnet-rdf.princeton.edu/id/03142912 http://wordnet-
rdf.princeton.edu/ontology#_hypernym http://wordnet-
rdf.princeton.edu/id/03081021
http://wordnet-rdf.princeton.edu/id/10722385 http://wordnet-
rdf.princeton.edu/ontology#_hypernym http://wordnet-
rdf.princeton.edu/id/10363913
http://wordnet-rdf.princeton.edu/id/11705921 http://wordnet-

```
rdf.princeton.edu/ontology#_member_meronym http://wordnet-
rdf.princeton.edu/id/11706325
http://wordnet-rdf.princeton.edu/id/02431337 http://wordnet-
rdf.princeton.edu/ontology#_hypernym http://wordnet-
rdf.princeton.edu/id/02431122
http://wordnet-rdf.princeton.edu/id/02067941 http://wordnet-
rdf.princeton.edu/ontology#_member_meronym http://wordnet-
rdf.princeton.edu/id/02068408
```

1.1.1 PyTorch

In this course we will make use of the *PyTorch* machine learning package, which provides a large array of convenient functions and tools to implement and run machine learning experiments. The core data structure in PyTorch is the *tensor*. Taking a broad definition of the tensor, it is used in PyTorch to store almost everything from scalars and vectors to matrices and higher-order tensors.

Much of your time will be spend working with this package.

Run the cells below to install the PyTorch package in your Python environment and to set a manual seed

[ ]:
```
# install pytorch
%pip install torch
```

```
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: torch in /home/samanu/.local/lib/python3.10/site-
packages (1.11.0+cu113)
Requirement already satisfied: typing-extensions in
/home/samanu/.local/lib/python3.10/site-packages (from torch) (4.4.0)
Note: you may need to restart the kernel to use updated packages.
```

[ ]:
```
import torch
import torch.nn as nn

seed = 42
torch.manual_seed(seed)  # allow for reproducability

import torch

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")
```

```
Using device: cuda
```

1.2 Data Preparation

Knowledge graphs are an extension of *labeled digraphs* and, as such, have labeled arcs and nodes. In case of the arcs, these labels represent relationship types, which typically occur more than once. In other words, arcs with the same label convey the same relationship. The situation is more complex for the nodes: some of the nodes represent *entities* (unique 'things', tangible or otherwise) which

are labeled using the *Universal Resource Identifier* (URI). Examples of entities are people, objects, concepts, and dreams. Other nodes represent *literals* (raw data values), such as numbers, dates, and strings.

Dealing with these different labels can be challenging. It is therefore common to simplify the knowledge graph to a regular digraph. This is done by first enumerating all unique labels, and by then re-encoding the graph using these numbers. This is done separately for the nodes and the relations.

1.2.1 Task 1: Encode the graph using incides

Write a procedure to transform the graph to an integer-encoding. For example, given a mapping `'maried' = 0`, `'age' = 5`, `'25' = 66`,`'john' = 43`, and `'kate' = 187`, the triples `(john, married, kate)` and `(john, age, 25)` should become `[43, 0, 187]` and `[43, 5, 66]`. Ensure that the encoding of the training set corresponds with that of the test set. The result should be two tensors and four maps.

[ ]:
```python
def encode_graph(
        graph: rdflib.Graph,
        common_entity_map=dict[rdflib.URIRef],
        common_relation_map=dict[rdflib.URIRef]
) -> list[list[int]]:
    """
    The encoding for an subject, predicate or entity will be the length
        of the corresponding entity_map at the moment of discovery.

    Args:
        graph (rdflib.Graph): The RDF graph to be encoded.
        common_entity_map (dict): A mapping of entities to integer indices.
        common_relation_map (dict): A mapping of relations to integer indices.

    Returns:
        graph_encoded (list): List of encoded triples as lists of integer␣
↪indices.
        common_entity_map (dict): Updated mapping of entities to integer␣
↪indices.
        common_relation_map (dict): Updated mapping of relations to integer␣
↪indices.
    """
    graph_encoded = []

    for subj, rel, obj in graph:
        # set_default() gets current value or size of entity_map
        # method also has leteral effect on entity_map and it's size can␣
↪increase
        subj_idx = common_entity_map.setdefault(subj, len(common_entity_map))
        rel_idx = common_relation_map.setdefault(rel, len(common_relation_map))
        obj_idx = common_entity_map.setdefault(obj, len(common_entity_map))
```

```
        graph_encoded.append([subj_idx, rel_idx, obj_idx])

    return graph_encoded, common_entity_map, common_relation_map

# instead of going with 4 maps, we decided to work with 2 maps for entities and␣
 ↪relations
# common_entity maps will be passed through the encoding of train and test data␣
 ↪in a cumulative way
# it will contain the indices for all entities and relations across train and␣
 ↪test
common_entity_map = {}
common_relation_map = {}
encoded_triples_train, common_entity_map, common_relation_map = encode_graph(
    graph=g_train,
    common_entity_map=common_entity_map,
    common_relation_map=common_relation_map
)

encoded_triples_test, common_entity_map, common_relation_map = encode_graph(
    graph=g_test,
    common_entity_map=common_entity_map,
    common_relation_map=common_relation_map
)

# convert the torch.Tensor
data_train = torch.LongTensor(encoded_triples_train)
data_test = torch.LongTensor(encoded_triples_test)

# test
assert len(data_train) == len(g_train)
assert len(data_test) == len(g_test)

print('TRAIN:', end='\n')
for triple in data_train[:10]:
    print(triple, end='\n')

print('TEST:', end='\n')
for triple in data_test[:10]:
    print(triple, end='\n')
```

```
TRAIN:
tensor([0, 0, 1])
tensor([2, 1, 3])
tensor([4, 1, 5])
tensor([6, 1, 7])
tensor([8, 0, 9])
```

```
tensor([10,  2, 11])
tensor([12,  3, 13])
tensor([14,  1, 15])
tensor([16,  1, 17])
tensor([18,  2, 19])
TEST:
tensor([19789,     1,   358])
tensor([22464,     1,  5869])
tensor([24002,     3, 39855])
tensor([33112,     1,    65])
tensor([ 5120,     0, 26714])
tensor([40559,     1, 22923])
tensor([ 1956,     1, 27746])
tensor([36355,     2, 23383])
tensor([40560,     1, 15224])
tensor([17748,     2, 32895])
```

1.3 A simple triple scoring model

Under the hood, a typical link prediction workflow defines a mathematical function f that takes a link as input and then returns the score of that link, or *triple*. The higher the score, the more likely the link is thought to exist. Given a set of candidate links, we can therefore compute all their scores and rank them in descending order. The top-k links are then the most likely to exist according to our model. Of course, to get reliable scores our model first needs to learn how to recognize possible valid links.

There are various ways for a link prediction model to learn how to distinguish between valid and invalid links. Some models use a parameterised scoring function with a set of weights that it updates every epoch, while treating the links' vector representations as immutable objects that have been initialized before training, as is common with traditional machine learning. Other methods, such as the one you will implement today, approach the problem in a different way, by optimizing the links' vector representations themselves. In a sense, the vector representations act as their own weights.

1.3.1 DistMult

Around 2015, a group of researchers of the Cornell University in the USA published a paper [1] in which they introduced a simple bilinear triple scoring function which performed surprisingly well on link prediction tasks. This scoring function, called *DistMult*, is defined as

$$y_{e1}^T M_r y_{e2} \qquad \text{(formula 2 in [1])}$$

with y_i the internal representation belonging to node i, and with M_r the diagonal matrix representation for relationship type r. Both these representations are updated each iteration by optimizing for a strong separation between positive and negative samples.

[1] Yang, B., Yih, S. W. T., He, X., Gao, J., & Deng, L. (2015). Embedding Entities and Relations for Learning and Inference in Knowledge Bases. In Proceedings of the International Conference on Learning Representations (ICLR) 2015.

1.3.2 Task 2: Implement DistMult

Implement DistMult as a subclass of the PyTorch `nn.Module` class. Specifically, this entails writing the initialisation function `__init__()` and a forward function `forward()`. Make it possible to specify the size of the dimensions used for the various vector representations. Test your model afterwards on a small subset of the training set.

[ ]:
```python
class DistMult(nn.Module):
    def __init__(self, num_entities: int, num_relations: int, embedding_dim:␣
 ↪int = 50):
        """ DistMult
        """
        super().__init__( )
        self.subj_emb = nn.Embedding(num_entities, embedding_dim).to(device)
        self.rel_emb = nn.Embedding(num_relations, embedding_dim).to(device)
        self.obj_emb = nn.Embedding(num_entities, embedding_dim).to(device)

    def forward(self, X: torch.Tensor):
        """ Forward pass
        Args:
            def forward(self, X: torch.Tensor): (torch.Tensor): Batch of␣
 ↪triplets (subject, relation, object)
        Returns:
            torch.Tensor: Scores for each triplet
        """
        # Extract subject, relation, and object indices
        subj_idx = X[:, 0]
        rel_idx = X[:, 1]
        obj_idx = X[:, 2]

        # Lookup embeddings for subject, relation, and object
        subj_emb = self.subj_emb(subj_idx)
        rel_emb = self.rel_emb(rel_idx)
        obj_emb = self.obj_emb(obj_idx)

        # # Compute scores using DistMult scoring function
        scores = torch.sum(subj_emb * rel_emb * obj_emb, dim=1)

        return scores
```

Run the following cell to initialize and test your implementation

[ ]:
```python
num_entities = len(common_entity_map)
num_relations = len(common_relation_map)
print(f"num_entities = {num_entities}")
print(f"num_relations = {num_relations}")
```

```
model = DistMult(num_entities=num_entities, num_relations=num_relations,␣
 ↪embedding_dim=50).cuda()

samples = data_train[:10].to(device)
out = model(samples)

print(f"forward out = {out}")
```

```
num_entities = 40768
num_relations = 11
forward out = tensor([  0.9353,  -3.3234,   2.3821,   2.3107,  -8.3056,
-2.2252, -19.7807,
          3.7478,   5.5972,  11.5480], device='cuda:0', grad_fn=<SumBackward1>)
```

1.4 Negative Sampling

A common approach for training link prediction models is to present the model with a set of positive and negative samples. Here, the positive samples are links that are known to exist in the graph, whereas the negative samples are links that are known never to exist in the graph. In practice, the set of positive samples is often just a subset of the graph, while the set of negative samples is generated by corrupting the existing links using some strategy. This is called *negative sampling*. By labeling the positive and negative samples accordingly, and by updating the weights on how these samples are scored, link prediction models learn to rank candidate links accurately.

1.4.1 Task 3a: Corrupting triples

Write a procedure to generate negative samples. The procedure should return a new tensor with non-existing triples. Use Python comments to explain the strategy that you decided on, and test your procedure on a small subset of the data afterwards.

[ ]:
```
def corrupt_triples(batch: torch.Tensor, full_data: torch.Tensor, num_entities:
 ↪int) -> torch.Tensor:
    corrupt_batch = []

    existing_triples = {(h.item(), r.item(), t.item()) for h, r, t in full_data}

    for link_to_corrupt in batch:
        # Randomly choose whether to corrupt the head or tail entity
        corrupt_head = torch.rand(1).item() > 0.5

        h, r, t = link_to_corrupt.tolist()

        # Randomly choose whether to corrupt the head or tail entity
        corrupt_head = torch.rand(1).item() > 0.5

        # Corrupt the chosen entity until a non-existing triple is obtained
        while (h, r, t) in existing_triples:
            if corrupt_head:
```

```
                h = torch.randint(num_entities, size=(1,)).item()
            else:
                t = torch.randint(num_entities, size=(1,)).item()

        assert (h, r, t) not in existing_triples
        corrupt_batch.append([h, r, t])

    return torch.LongTensor(corrupt_batch)
```

Run the following cell to test your procedure.

[ ]:
```
samples = data_train[:10]
print(f"original data:\n{samples}\n")

samples_corrupted = corrupt_triples(data_train[:10], data_train, num_entities)
print(f"corrupted data:\n{samples_corrupted}")

# test
all_triples = {(h.item(), r.item(), t.item()) for h, r, t in data_train}
for corrupt_triple in samples_corrupted:
    h, r, t = corrupt_triple.tolist()
    assert (h, r, t) not in all_triples
```

```
original data:
tensor([[ 0,  0,  1],
        [ 2,  1,  3],
        [ 4,  1,  5],
        [ 6,  1,  7],
        [ 8,  0,  9],
        [10,  2, 11],
        [12,  3, 13],
        [14,  1, 15],
        [16,  1, 17],
        [18,  2, 19]])

corrupted data:
tensor([[    0,     0,  5746],
        [33933,     1,     3],
        [    4,     1, 16898],
        [35471,     1,     7],
        [    8,     0, 29712],
        [   10,     2, 26734],
        [ 9751,     3,    13],
        [36267,     1,    15],
        [31833,     1,    17],
        [37356,     2,    19]])
```

1.5 Mini Batching

Real-world graphs are often quite large, which translates in a large number of links. It is, for example, not uncommon for a graph to have many millions of links. Just think about how many people are subscribed to Facebook, how many others they are connected to, and how many attributes an average user has associated with their account. Learning over so many links in a single go is often impossible, especially if we want to speed up our computations using a GPU, which has a limited amount of on-device memory.

Mini batching is a technique that enables you to learn on partitions of the data. This is typically achieved by splitting the data into roughly even parts, and by updating the model after each part. Next to alleviating scalibity issues, updating the model after seeing a subset of the data has the additional benefit of creating a smoother convergence, since any sudden changes in gradient by individual samples is compensated by other samples in the batch.

1.5.1 Task 3b: Creating batches

Write a procedure to generate mini batches of triples. The procedure should return a list of batches. Take into account the overhead that copying batches to GPU has by distributing the triples in some sensible way. The preferred batch size must be a parameter.

[ ]:
```python
def mk_batches(full_data: torch.Tensor, batch_size: int) -> list:
    # permute the data
    shuffled_data = full_data[torch.randperm(full_data.size(0))]

    # calculate the number of batches
    num_batches = (shuffled_data.size(0) + batch_size - 1) // batch_size

    # initialize an empty list to store batches
    batches = []

    # create mini-batches
    for i in range(num_batches):
        start_idx = i * batch_size
        end_idx = min((i + 1) * batch_size, shuffled_data.size(0))

        # allow only for complete batches
        if end_idx - start_idx == batch_size:
            batch_data = shuffled_data[start_idx:end_idx]
            batches.append(batch_data)

    return batches
```

Run the following cell to test your function

[ ]:
```python
for i, batch in enumerate(iterable = mk_batches(data_train, len(data_train) //␣
↪3), start= 1):
    print(f"batch {i}: {batch}")
```

```
batch 1: tensor([[ 4512,     7, 12447],
        [  786,     9, 11585],
        [38808,     2, 17845],
        …,
        [28853,     1,  2030],
        [20743,     1,   269],
        [ 3048,     0,  8861]])
batch 2: tensor([[ 9699,     0, 20065],
        [15030,     1, 16302],
        [25364,     1, 28290],
        …,
        [ 3465,     1, 24327],
        [ 8545,     3, 33560],
        [23176,     0, 27539]])
batch 3: tensor([[10679,     1,  6800],
        [11624,     0,  8140],
        [14435,     0, 22738],
        …,
        [  929,     0,  8936],
        [10977,     0,  3035],
        [ 1282,     1,  1794]])
```

1.6 Training and testing

Training and testing are two important steps in the experimental workflow. Here, it is important to remember that the test data must be used exactly once, for testing, and that the test run must use a predetermined set of hyperparameter values. If this were not the case, then you are simply optimizing for improved performance on the test split rather than on the entire dataset.

While left out here, it is also often good practice to create a validation set. The validation set can be used for hyperparameter optimization by training the model with various different hyperparameter values and by evaluating the performance on the validation set.

1.6.1 Task 4a: The training loop

Write a procedure to train the model. Specifically, create a loop that passes the entire training set through the model every epoch, while computing the loss and updating the weights after each batch. Ensure that your batches and negative samples get randomized or regenerated each epoch. Use the Adam optimizer and the BCE with logits loss.

[ ]:
```python
# set hyperparameters
learning_rate = 0.01
num_epoch = 10
batch_size = 2048

# set optimizer and loss function
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
loss_function = torch.nn.BCEWithLogitsLoss()
```

```
num_train = data_train.shape[0]
for epoch in range(1, num_epoch+1):
    print(f'Epoch {epoch:3d} - ', end='')

    # create batches
    batches = mk_batches(data_train, batch_size)
    num_batches = len(batches)

    batch_loss_tensor = torch.zeros(num_batches, dtype=torch.float32)
    for batch_id, batch in enumerate(batches):
        data_batch = batch
        num_samples = data_batch.shape[0]

        # create negative samples by randomly assigning random node indices in␣
 ↪the object position
        data_batch_corrupt = corrupt_triples(batch=batch, full_data=data_train,␣
 ↪num_entities=num_entities)

        # create labels; positive samples are 1, negative 0
        Y_positive = torch.ones(num_samples, dtype=torch.float32)
        Y_negative = torch.zeros(num_samples, dtype=torch.float32)
        Y = torch.cat([Y_positive, Y_negative]).to(device)

        # allow model parameters to be learned
        model.train()

        # compute scores for positive and negative triples
        current_input = torch.cat([data_batch, data_batch_corrupt]).to(device)
        Y_hat = model(current_input).to(device)

        # compute loss
        batch_loss = loss_function(Y_hat, Y)

        # Zero gradients, perform a backward pass, and update the weights.
        optimizer.zero_grad()
        batch_loss.backward()
        optimizer.step()

        batch_loss = float(batch_loss)  # release memory of computation graph
        batch_loss_tensor[batch_id] = batch_loss

    print(f'loss on train set: {batch_loss_tensor.mean():0.4f}')
```

```
Epoch   1 - loss on train set: 2.2865
Epoch   2 - loss on train set: 1.3978
Epoch   3 - loss on train set: 0.9876
Epoch   4 - loss on train set: 0.7928
```

```
Epoch   5 - loss on train set: 0.7020
Epoch   6 - loss on train set: 0.6570
Epoch   7 - loss on train set: 0.6260
Epoch   8 - loss on train set: 0.6061
Epoch   9 - loss on train set: 0.5862
Epoch  10 - loss on train set: 0.5694
```

1.6.2 Task 4b: The test loop

Write a procedure to test the now-trained model. Specifically, create a loop that passes the entire test set through the model every epoch, while computing the loss after each batch. Ensure that the weights of your model are frozen during testing.

[ ]:
```python
num_test = data_test.shape[0]

batch_size = 32
batches = mk_batches(data_test, batch_size)
num_batches = len(batches)

# we use num_samples * 2 to test if the score goes to 1
# we have chosed to always corrupt the same size as the batch
k_values = [1, 3, 5, 10, batch_size, batch_size * 2]
hits_at_k_accumulator = {k: 0.0 for k in k_values}

batch_loss_tensor = torch.zeros(num_batches, dtype=torch.float32)
for batch_id, batch in enumerate(batches):
    data_batch = batch
    num_samples = data_batch.shape[0]
    assert num_samples == batch_size

    # create negative samples by randomly assigning random node indices in the␣
 ↪object position
    data_batch_corrupt = corrupt_triples(batch=batch, full_data=data_train,␣
 ↪num_entities=num_entities)

    # create labels; positive samples are 1, negative 0
    Y_positive = torch.ones(num_samples, dtype=torch.float32)
    Y_negative = torch.zeros(num_samples, dtype=torch.float32)
    Y = torch.cat([Y_positive, Y_negative]).to(device)

    # freeze model parameters for evaluation
    model.eval()

    # compute scores for positive and negative triples
    current_input = torch.cat([data_batch, data_batch_corrupt]).to(device)
    Y_hat = model(current_input).to(device)

    # compute loss
```

```
    batch_loss = loss_function(Y_hat, Y)

    batch_loss = float(batch_loss)  # release memory of computation graph
    batch_loss_tensor[batch_id] = batch_loss

    # compute hits@k (see task 5)
    hits_at_k_for_current_batch = compute_hits_at_k(Y_hat, Y,␣
 ↪first_positive_samples=batch_size, k_values=k_values)

    # Accumulate hits@k scores across batches
    for k, v in hits_at_k_for_current_batch.items():
        hits_at_k_accumulator[k] += v

# Calculate the average hits@k scores for all batches
average_hits_at_k = {k: v / num_batches for k, v in hits_at_k_accumulator.
 ↪items()}

print(f'loss on test set: {batch_loss_tensor.mean():0.4f}')

for k,v in average_hits_at_k.items():
    print(f'Average across all batches hits@{k}: {v:0.4f}')
```

```
loss on test set: 0.8110
Average across all batches hits@1: 0.0312
Average across all batches hits@3: 0.0493
Average across all batches hits@5: 0.0767
Average across all batches hits@10: 0.1562
Average across all batches hits@32: 0.5000
Average across all batches hits@64: 1.0000
```

1.6.3 Evaluation

Two different metrics are commonly used to evaluate link prediction models: the *mean reciprocal rank* (MRR) and *hits@k*. The MRR equals the arithmetic mean of reciprocal ranks, and is defined as

$$MRR = \frac{1}{|\mathcal{T}|} \sum_{t \in \mathcal{T}} r(t)^{-1}$$

with $\mathcal{T}$ the set of positive triples and $r(t)$ the rank of triple t. The MRR returns a score in $(0, 1]$ where higher is better.

The *hits@k* equals the fraction of positive samples that appear in the first k entries of the sorted rank list. The metric is similar to the MRR, yet cheaper to compute and more sensitive to irregularities. The *hits@k* is formaly defined as

$$\text{hits@k} = \frac{1}{|\mathcal{T}|} \sum_{t \in \mathcal{T}} 1 \text{ iff } r(t) \leq k$$

Common values for k are 1, 3, 5, and 10. The returned score lie between $(0, 1]$ where higher is better.

1.6.4 Task 5: Evaluate the model

Write a procedure to calculate the *hits@k* metric and update task 4b to output the average *hits@k* score for $k \in \{1, 3, 10\}$. Ensure that the score equals 1.0 if all top-k entries are positive samples. Be aware that you will need to rerun the test loop.

[ ]:
```
def compute_hits_at_k(Y_hat: torch.Tensor, Y: torch.Tensor,␣
↪first_positive_samples: int, k_values=[1, 3, 10]) -> dict:

    hits_at_k_scores = {}

    # Transfer tensors to CPU for processing (if not already)
    Y_hat = Y_hat.cpu()
    positive_samples = Y_hat[:first_positive_samples]
    Y = Y.cpu()

    for k in k_values:
        # get the top-k predictions for each sample
        top_k_preds = torch.topk(Y_hat, k, dim=0, largest=False)[1]
        positive_preds = torch.topk(positive_samples, first_positive_samples,␣
↪dim=0, largest=False)[1]

        # compute the percentage
        hits_at_k = torch.sum(torch.isin(positive_preds, top_k_preds)).item() /␣
↪first_positive_samples
        hits_at_k_scores[k] = hits_at_k

    return hits_at_k_scores
```

1.7 Deliverable

Once you and your team member are satisfied with the results, you are to rerun the notebook from scratch and to export it to PDF format. This cleans up the output and makes it easier for us to provide feedback. Specifically, first select Restart kernel and run all cells followed by Save and export Notebook as -> PDF.

Please rename the PDF file to `ML4G-PA1_GROUP6.pdf` and submit the file before noon the next Monday