Learning how to use GNN

To get started, you need to install pytorch geometric:
`https://pytorch-geometric.readthedocs.io/en/latest/install/installation.html`.
You can then have a look at the tutorial:
`https://pytorch-geometric.readthedocs.io/en/latest/get_started/introduction.html`.
Check in particular sections:

- Data Handling of Graphs
- Data Transforms
- Learning Methods on Graphs

1. Getting started: data preparation for node classification
   (a) Load the toy dataset ToyFriendship.graphml from the class website, using networkx. Plot the network to have a quick view, check the attributes (`G.nodes(data=True)`). We consider that we know the preferences of the students among sports/music/science, and the club they belong to.
   (b) Convert from networkx to pytorch geometric using `from_networkx` function. Be careful, due to some bug, you first need to do `G.graph={}` on your networkx graph. In the function, use `group_node_attrs=["like_sports","like_music","like_science"]` to load only those attributes as `x` .
   (c) Check what is inside this object. You should find the edges `edge_index` , the node features `x` , etc.
   (d) Encode the class (`club`) using sklearn `LabelEncoder` , e.g.,

```
encoder = LabelEncoder()
integer_labels = encoder.fit_transform(data.club)
target_tensor = torch.tensor(integer_labels, dtype=torch.long)
data.y = target_tensor
data.num_classes = len(set(data.club))
```

   (e) Let's consider that for some of the students, we don't know their preferences, but we want to train a model to guess the club they belong to. For instance, we can imagine new students to whom we want to recommend a club. So we want to guess the club class from the `like` attributes, but for students for which we don't have the like attribute. Without graph, this is not possible.
   (f) We need to hide the `like` information for some of the nodes. You can do it with a mask, with something like:

```
num_nodes = data.num_nodes
train_ratio = 0.80  # 80% of nodes for training

# Randomly creating a mask
mask = torch.rand(num_nodes) < train_ratio
data.train_mask = mask
data.test_mask = ~data.train_mask

# remove the attributes for the nodes that are not in the training set
temp = torch.zeros((num_nodes, 3), dtype=torch.float)
temp[data.train_mask] = data.x[data.train_mask]
data.x = temp
```

2. Predict using a GCN
   (a) Build your first GCN, with a single layer. It should solve a classification problem, with 3 classes.
   (b) Your evaluation should be only on the test set, i.e., something like:

```
pred = model(data).argmax(dim=1)
correct = (pred[data.test_mask] == data.y[data.test_mask]).sum()
acc = int(correct) / int(data.test_mask.sum())
print(f'Accuracy: {acc:.4f}')
```

   (c) Check the `add_self_loops` attributes of the conv layer, and think of its meaning.
   (d) Print the targets and the predictions for all the nodes
   (e) Plot a confusion matrix, e.g.,

```
from sklearn.metrics import confusion_matrix
import seaborn as sns
cm = confusion_matrix(data.y[data.test_mask],pred[data.test_mask])
sns.heatmap(cm, annot=True, fmt='g')
```

   (f) Print the weights of the GCN layer and interpret them (if you have a good accuracy...)

3. Using a GAT
   (a) The problem is the same, but replace the `GCNConv` layer with a `GATv2Conv` layer, with a single head of attention, for now.
   (b) Do avoid overfit, you might want to add a `F.dropout` before the layer
   (c) By using the attribute `return_attention_weights=True)` when calling the GAT layer for prediction, you can access the attention computed by each neighbor on each other. Check what is in what is returned
   (d) You can plot the attention computed using something like: (you might need to adapt the code to your setting, of course)

```
predict,attention = model.gat1(data.x,data.edge_index,return_attention_weights=True)

attention_w=attention[1].detach().numpy()
#get the edges
edges= attention[0].detach().numpy()

#convert the edges to a dataframe
df=pd.DataFrame(edges.T,columns=["source","target"])
df["attention"]=attention_w
cmap = cm.get_cmap('coolwarm')
edge_colors = cmap(df['attention'].tolist())
#plot as a networkx graph with the attention weights as colors
G=nx.from_pandas_edgelist(df,"source","target",edge_attr="attention")
plt.figure(figsize=(10,10))

#add the club attribute to the nodes
nx.set_node_attributes(G,dict(zip(range(len(data.club)),data.club)),"club")
colors = []
for node in G.nodes(data=True):
    if node[1]['club'] == 'Sports':
        colors.append('red')
    elif node[1]['club'] == 'Music':
        colors.append('green')
    else:
        colors.append('blue')
nx.draw_networkx(G,with_labels=True,edge_color=edge_colors,width=1,node_size=100,node_color=
     colors)
```

4. Predicting edges using a VGAE
   (a) This time, we want to predict edges. Start back from the original network, without hidden information. Build a train_test split using the `train_test_split_edges` function from `torch_geometric.utils`. You don't need a validation set.
   (b) Build your `Encoder`. As seen in class, it should be something like:

```
class Encoder(torch.nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.conv1 = GCNConv(in_channels, 2*out_channels)
        self.conv_mu = GCNConv(2*out_channels, out_channels)
        self.conv_logstd = GCNConv(2*out_channels, out_channels)

    def forward(self, x, edge_index):
        x = self.conv1(x, edge_index).relu()
        return self.conv_mu(x, edge_index), self.conv_logstd(x, edge_index)
```

   (c) Initialize your model using `VGAE` from `torch_geometric.nn`.
   (d) Evaluate your model. Be careful to use training data for training and testing data for testing :) Something like:

```
    z = model.encode(data.x, data.train_pos_edge_index)
    return model.test(z, data.test_pos_edge_index, data.test_neg_edge_index)
```

(e) Evaluate the result of your test, i.e., with AUC and AP (Average Precision)

(f) Check that you are able to reconstruct the original graph, by applying the dot product between node vectors. You can do it with something like:

```
z = model.encode(data.x, data.train_pos_edge_index)
Ahat = torch.sigmoid(z @ z.T)
```