Master Informatique, parcours MALIA & MIASHS

Carnets de note Python pour le cours de Network Analysis for Information Retrieval

Julien Velcin, laboratoire ERIC, Université Lyon 2

Merci à **Jacques Fize**, post-doc sur le projet POIVRE en 2022, pour avoir créé la première version de ce notebook, version que j'ai ensuite fait évoluer pour le cours.

Ce carnet permet de charger un jeu de données bien connu dans la littérature (CORA), d'en faire des visualisations puis de lancer des algorithmes de classification supervisée (classiques et GNNs).

Requirements

Pour la partie machine learning :

- pytorch
- torch geometric

Pour la gestion des graphes :

- networkx
- bokeh

Autre :

- tqdm
- numpy

In [3]:
```
import os.path as osp

import torch_geometric.transforms as T
from torch_geometric.datasets import Planetoid
from torch_geometric.nn import GCNConv, GATConv, GATv2Conv

import torch
import torch.nn.functional as F

from networkx import Graph, draw, set_node_attributes, spring_layout
from bokeh.io import output_notebook, show, save
from bokeh.models import Range1d, Circle, ColumnDataSource, MultiLine
from bokeh.plotting import figure
from bokeh.plotting import from_networkx
```

```
import numpy as np
from tqdm import tqdm
```

Chargement des données

La classe `Planetoid` permet de charger des jeux de données connus dans l'état de l'art. D'autres classes sont disponibles (ici) pour charger différents datasets. Dans l'exemple ci-dessous, on charge le graphe Cora qui contient des articles scientifiques.

In [4]:
```
dataset = 'Cora'
path = osp.join('data', dataset)
dataset = Planetoid(path, dataset, split="full")
data = dataset[0] # un seul graphe dans le dateset
data
```

- `x` correspond à la matrice contenant les *features* de chaque sommet du graphe

```
tensor([[0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.],
        ...,
        [0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.]])
```

- `edge_index` indique les connexions entre les sommets du graphe:

```
tensor([[   0,    0,    0,  ..., 2707, 2707, 2707],
        [ 633, 1862, 2582,  ...,  598, 1473, 2706]])
```

- `y` correspond aux labels associés pour chaque noeud :

`tensor([3, 4, 4,  ..., 3, 3, 3])`

- `train_mask`, `val_mask` et `test_mask` sont des masques booléens pour indiquer quels sommets seront utilisée pour l'entrainement et l'évaluation du modèle (GNN):

`tensor([ True,  True,  True,  ..., False, False, False])`

In [5]:
```
data
```

In [6]:
```
# Je laisse ça là, au cas où ça peut être utile par la suite :

#Différentes méthodes sont disponibles pour modifier les données. Ici, on utilise `
#transform = T.Compose([
#    T.RandomNodeSplit(num_val=500, num_test=500),
#    T.TargetIndegree(),
#])
```

```
#data = transform(data)
```

Distribution des données sur les 7 classes en fonction du sous-ensemble (train, validation, test)

In [7]:
```
unique_values_a, counts_a = np.unique(data.y[data.train_mask].tolist(), return_c
unique_values_b, counts_b = np.unique(data.y[data.test_mask].tolist(), return_count
unique_values_c, counts_c = np.unique(data.y[data.val_mask].tolist(), return_counts

print(f"Jeu d'entrainement ({len(data.y[data.train_mask].tolist())}) : " + " ".join
print(f"Jeu de validation ({len(data.y[data.val_mask].tolist())}) : " + " ".join([f
print(f"Jeu de test ({len(data.y[data.test_mask].tolist())}) : " + " ".join([f"{val
```

Visualisation des données

On utilise la librairie networkx pour réaliser quelques visualisations.

In [8]:
```
G = Graph()

n_i = data.edge_index[0,:].tolist()
n_j = data.edge_index[1,:].tolist()

num_vertices = len(data.y.tolist())
all_vertices = [(i, {"class": data.y.tolist()[i]}) for i in range(num_vertices)]

G.add_nodes_from(all_vertices)
```

In [9]:
```
all_edges = [(n_i[i], n_j[i]) for i in range(len(n_i))]

G.add_edges_from(all_edges)
```

In [10]:
```
all_edges[0:10]
```

Calcul des degrés et construction d'un sous-graphe plus dense pour la visualisation.

In [11]:
```
degrees = [v for k,v in G.degree()]

high_degrees = [n_i for n_i in range(num_vertices) if degrees[n_i]>30]
subG = G.subgraph(high_degrees)
```

In [12]:
```
draw(subG, with_labels=True)
```

La librairie de base n'étant pas suffisamment robuste, on emploie ici la librairie *bokeh* qui permet une exploration interactive du graphe (ici tout le graphe).

In [13]:
```
#see: https://melaniewalsh.github.io/Intro-Cultural-Analytics/06-Network-Analysi

output_notebook()

title = 'My network'
```

```
HOVER_TOOLTIPS = [("id", "@index"), ("class", "@class")]

plot = figure(tooltips = HOVER_TOOLTIPS,active_scroll='wheel_zoom', title=title, wi

dico_color = {
        0: "skyblue",
        1: "green",
        2: "blue",
        3: "red",
        4: "yellow",
        5: "dark",
        6: "pink"
    }

node_colors = {}
for i in range(num_vertices):
    node_colors[i] = dico_color[data.y.tolist()[i]]

set_node_attributes(G, node_colors, "node_color")

network_graph = from_networkx(G, spring_layout, scale=10, center=(0, 0))

network_graph.node_renderer.glyph = Circle(size=15, fill_color="node_color") #'skyb
network_graph.edge_renderer.glyph = MultiLine(line_alpha=0.5, line_width=1)

plot.renderers.append(network_graph)
plot.axis.visible = False

show(plot)
#save(plot, filename=f"{title}.html")
```

Classification supervisée

Préparatin des données en trois sous-ensemble.

In [14]:
```
numx_train, num_feat = data.x[data.train_mask].shape
print(f"Dim train: ({numx_train},{num_feat})")
numx_val, _ = data.x[data.val_mask].shape
print(f"Dim val: ({numx_val},{num_feat})")
numx_test, _ = data.x[data.test_mask].shape
print(f"Dim test: ({numx_test},{num_feat})")
```

Algorithmes classiques de machine learning

In [15]:
```
# différents classifieurs
from sklearn.linear_model import LogisticRegression
from sklearn.linear_model import RidgeClassifier
from sklearn.svm import SVC
from xgboost import XGBClassifier
from sklearn.neural_network import MLPClassifier
```

In [16]:
```
def print_accuracy(nom_algo, truth_train, pred_train, truth_test, pred_test):
    print(nom_algo + " : ")
    acc_app = np.sum(pred_train == truth_train) / float(len(pred_train))
    print(f"  - réussite (accuracy) apparente : {acc_app:.1%}")
    acc_gen = np.sum(pred_test == truth_test) / float(len(pred_test))
    print(f"  - réussite (accuracy) en généralisation : {acc_gen:.1%}")
```

In [17]:
```
lr = LogisticRegression()

lr.fit(data.x[data.train_mask], data.y[data.train_mask])
pred_train_lr = lr.predict(data.x[data.train_mask])
pred_test_lr = lr.predict(data.x[data.test_mask])

print_accuracy("lr", data.y[data.train_mask].tolist(), pred_train_lr, data.y[data.t
```

In [18]:
```
#print("Ce qui est prédit : " + str(pred_y_lr[0:20]))
#print("La vérité : " + str(data.y[data.test_mask].tolist()[0:20]))
```

In [19]:
```
svc_poly2 = SVC(kernel="poly", degree=2)

svc_poly2.fit(data.x[data.train_mask], data.y[data.train_mask])

pred_train_svc_poly2 = svc_poly2.predict(data.x[data.train_mask])
pred_test_svc_poly2 = svc_poly2.predict(data.x[data.test_mask])

print_accuracy("SVC poly degré 2", data.y[data.train_mask].tolist(), pred_train_sv
```

In [20]:
```
mlp = MLPClassifier(solver='adam', alpha=1e-5, hidden_layer_sizes=(5), random_st

mlp.fit(data.x[data.train_mask], data.y[data.train_mask])

pred_train_mlp = mlp.predict(data.x[data.train_mask])
pred_test_mlp = mlp.predict(data.x[data.test_mask])

print_accuracy("MLP 1 couche cachée", data.y[data.train_mask].tolist(), pred_train_
```

Graph Neural Networks (GCN et GAT)

On définit une classe MyGCN de la librairie PyTorch avec plusieurs couches de convolution.

In [21]:
```
class MyGCN(torch.nn.Module):
    def __init__(self, d, n_feat):
        super().__init__()
        self.d = d
        # Initialisation des couches de convolutions
        self.conv1 = GCNConv(n_feat, 16)
        self.conv2 = GCNConv(16, dataset.num_classes)

    def forward(self):
        # Récupération des données
        x, edge_index, edge_attr = self.d, data.edge_index, data.edge_attr
        # Premiere convolution
```

```
        x = F.dropout(x, training=self.training)
        x = F.elu(self.conv1(x, edge_index, edge_attr))
        # Deuxième convolution
        x = F.dropout(x, training=self.training)
        x = self.conv2(x, edge_index, edge_attr)
        # Softmax
        return F.log_softmax(x, dim=1)
```

La classe suivante gère l'entraînement du modèle

In [22]:
```
class Training():

    def __init__(self, m, o):
        self.model = m
        self.optim = o

    def train(self, nb_epochs=50):
        progress_bar = tqdm(range(nb_epochs))
        for epoch in progress_bar:
            train_acc, test_acc = self.test()
            progress_bar.set_description(f'Epoch: {epoch:03d}, Train: {train_acc:.4
            self.model.train() ## mode "train"
            self.optim.zero_grad()
            F.nll_loss(self.model()[data.train_mask], data.y[data.train_mask]).back
            self.optim.step()

    def eval(self):
        self.model.eval()

    def forward(self):
        return self.model.forward()

    # retourne l'accuracy sur les données d'entraînement (train) et sur le test
    @torch.no_grad()
    def test(self):
        self.model.eval()  ## mode "eval" (pas de dropout)
        log_probs, accs = self.model(), []
        for _, mask in data('train_mask', 'test_mask'):
            pred = log_probs[mask].max(1)[1]
            acc = pred.eq(data.y[mask]).sum().item() / mask.sum().item()
            accs.append(acc)
        return accs

    @torch.no_grad()
    def print_eval(self, name):
        self.model.eval()  ## mode "eval" (pas de dropout)
        log_probs = self.model()
        pred_train = np.array(log_probs[data.train_mask].max(1)[1])
        pred_test = np.array(log_probs[data.test_mask].max(1)[1])
        print_accuracy(name, data.y[data.train_mask].tolist(), pred_train, data.y[d
```

In [23]:
```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model_GCN1, data = MyGCN(data.x, dataset.num_features).to(device), data.to(device)
optimizer_GCN1 = torch.optim.Adam(model_GCN1.parameters(), lr=0.01, weight_decay=5e
```

In [24]:
```
m1 = Training(model_GCN1, optimizer_GCN1)
m1.train(nb_epochs=200)
```

In [25]:
```
m1.print_eval("GCN with features")
```

In [26]:
```
num_data = data.x.shape[0]
data_random = torch.rand(num_data, dataset.num_features)
```

In [27]:
```
model_nofeatures = MyGCN(data_random, dataset.num_features).to(device)
optimizer_no_features = torch.optim.Adam(model_nofeatures.parameters(), lr=0.01, we
```

In [28]:
```
m2 = Training(model_nofeatures, optimizer_no_features)
m2.train()
m2.print_eval("GCN random features")
```

In [29]:
```
data_ohe = F.one_hot(torch.arange(0, num_data)).type('torch.FloatTensor')
model_ohe = MyGCN(data_ohe, num_data).to(device)
optimizer_ohe = torch.optim.Adam(model_ohe.parameters(), lr=0.01, weight_decay=5e-3
m3 = Training(model_ohe, optimizer_ohe)
m3.train()
m3.print_eval("GCN OHE")
```

Essaie avec un GAT (v1 puis v2)

In [30]:
```
class MyGAT(torch.nn.Module):
    def __init__(self, d, n_feat):
        super().__init__()
        self.d = d
        # Initialisation des couches de convolutions
        self.conv1 = GATConv(n_feat, 16)
        self.conv2 = GATConv(16, dataset.num_classes)

    def forward(self):
        # Récupération des données
        x, edge_index, edge_attr = self.d, data.edge_index, data.edge_attr
        # Premiere convolution
        x = F.dropout(x, training=self.training)
        x = F.elu(self.conv1(x, edge_index, edge_attr))
        # Deuxième convolution
        x = F.dropout(x, training=self.training)
        x = self.conv2(x, edge_index, edge_attr)
        # Softmax
        return F.log_softmax(x, dim=1)
```

In [31]:
```
modelGATv1, data = MyGAT(data.x, dataset.num_features).to(device), data.to(devic
optimizerGATv1 = torch.optim.Adam(modelGATv1.parameters(), lr=0.01, weight_decay=5e
m4 = Training(modelGATv1, optimizerGATv1)
m4.train()
m4.print_eval("GATv1")
```

In [32]:
```
class MyGATv2(torch.nn.Module):
    def __init__(self, d, n_feat):
        super().__init__()
        self.d = d
        # Initialisation des couches de convolutions
        self.conv1 = GATv2Conv(n_feat, 6, heads=4) # on essaie d'utiliser 4 têtes d
        self.conv2 = GATv2Conv(24, dataset.num_classes)

    def forward(self):
        # Récupération des données
        x, edge_index, edge_attr = self.d, data.edge_index, data.edge_attr
        # Premiere convolution
        x = F.dropout(x, training=self.training)
        x = F.elu(self.conv1(x, edge_index, edge_attr))
        # Deuxième convolution
        x = F.dropout(x, training=self.training)
        x = self.conv2(x, edge_index, edge_attr)
        # Softmax
        return F.log_softmax(x, dim=1)
```

In [33]:
```
modelGATv2, data = MyGATv2(data.x, dataset.num_features).to(device), data.to(dev
optimizerGATv2 = torch.optim.Adam(modelGATv2.parameters(), lr=0.01, weight_decay=5e
m5 = Training(modelGATv2, optimizerGATv2)
m5.train()
m5.print_eval("GATv2")
```

Une rapide conclusion

On constate que :

- les caractéristiques textuelles sont très importantes
- la prise en compte de la structure permet de gagner +10% environ

In [ ]: