cs224w colab 0

1. NetworkX Tutorial

- directed, undirected, multigraph 와 같은 다양한 그래프 타입들 제공
- Graph

```
import networkx as nx

G=nx.Graph() ## Undirected graph
H=nx.DiGraph() ## Directed graph

G.is_directed() ## 이것을 통해 directed graph 여부 확인
G.graph ## 이것을 통해 graph level attribute 확인

G.graph['Name']='Bar' ## graph level attribute 삽입
```

- Node

```
## 새로운 노드 하나 추가
G.add_node(0,feature=5,label=0)
node_0_attr=G.nodes[0] ## 노드 0의 속성 출력

## 노드 여러 개 추가
G.add_nodes_from([
  (1, {"feature": 1, "label": 1}),
  (2, {"feature": 2, "label": 2})
]) #(노드, dictionary 형태의 속성들)

for node in G.nodes(): ## 해당 graph에 존재하는 node들 반환
  print(node)


for node in G.nodes(data=True): ## data=True라고 할 경우 node들 속성 반환
  print(node)

# graph가 가진 노드들의 개수 반환
num_nodes = G.number_of_nodes()
print("G has {} nodes".format(num_nodes))
```

- Edge

```
## undirected graph에 새로운 edge 하나 추가
G.add_edge(0,1,weight=0.5) ## 이때 가중치는 0.5

## undirected graph에 여러개 edge 추가
G.add_edges_from([
  (1, 2, {"weight": 0.3}),
  (2, 0, {"weight": 0.1})
])

for edge in G.edges(): ## 해당 graph에 존재하는 edge들 반환
  print(edge)

## edge 속성 확인 - 이때 undirected graph라 (0,1)==(1,0)
edge_0_1_attr=G.edges[(0,1)]
edge_1_0_attr=G.edges[(1,0)]

# edge 개수 확인
num_edges = G.number_of_edges()

## directed graph에 새로운 edge 하나 추가
H.add_edge(0,1,weight=0.5)
```

```
## edge 속성 확인 - 이때 directed graph라 (1,0)에 대해 속성 출력하면 에러남
edge_0_1_attr=H.edges[(0,1)]
edge_1_0_attr=H.edges[(1,0)]
```

- Visualization
  - 간단한 방법

```
nx.draw(G,with_labels=True)
```

  - 다른 방법

```
import matplotlib.pyplot as plt

## 모든 node들에 대한 position - seed는 reporducibility를 위한 것
pos=nx.spring_layout(G,seed=7)

## node 그리기
nx.draw_networkx_nodes(G,pos,node_size=700)

## edge 그리기
nx.draw_networkx_edges(G,pos,edgelist=G.edges(),width=6,edge_color='b',style='dashed')

## node label
nx.draw_networkx_labels(G,pos,font_size=20)

## edge weight label
nx.draw_networkx_labels(G,pos,font_size=20)
edge_labels=nx.get_edge_attributes(G,'weight')
nx.draw_networkx_edge_labels(G,pos,edge_labels)

ax=plt.gca()
ax.margins(0.08)
plt.axis("off")
plt.tight_layout()
plt.show()
```

- Node degree와 neigbor 찾기

```
## node degree 반환
G.degree[node_id]

## node의 neighbor 반환
list(G.neighbors(node_id)) ## list형 반환

for neighbor in G.neighbors(node_id): ## dict_keyiterator로 반환한다
  print(neighbor)
```

- Other functionalities

```
num_nodes = 4

## 새로운 그래프 생성 후 directed graph로 지정
## path_graph(n)은 n개의 node + n-1개 edge 를 가진 linearly connected graph를 생성한다
G = nx.DiGraph(nx.path_graph(num_nodes))
nx.draw(G, with_labels = True)

# Get the PageRank
pr = nx.pagerank(G, alpha=0.8) ## alpha : 감폭 비율
pr
```

  - PageRank

    incoming link에 대한 structure을 기반으로 graph의 노드들 ranking을 계산한 것

2. Pytorch Geometric

- Dataset
  - 34의 karate 클럽 멤버들 각각을 하나의 노드, 멤버들이 사적으로 교류가 있었다면 해당 노드들을 엣지로 연결한 그래프 데이터 → features는 노드 feature 개수 (34명이니까 34 feature)
  - 각 노드는 4개의 커뮤니티 중 어떤 커뮤니티에 속해 있는지의 label을 가짐 → class

```
from torch_geometric.datasets import KarateClub

dataset=KarateClub()

## 그래프 개수
len(dataset)
## n-dimensional feature vector - 노드들이 가진 feature
dataset.num_features
## class 개수 - 노드들이 가진 label
dataset.num_classes


data=dataset[0] ## 그래프 가져오기

## 그래프에 대한 간단한 설명
print(data) ## 노드 개수, 엣지 개수, y, 훈련 노드 개수(train_mask)

# 그래프에 대한 properties
## 노드와 엣지 개수
data.num_nodes
data.node_edges
## edge들이 어떤 node를 연결하고 있는지 출력
data.edge_index.t()
## undirected graph인지 여부
data.is_undirected() ## undirected임
## 평균 node degree
2*data.num_edges/data.num_nodes
## 훈련 노드들 개수
data.train_mask.sum()
## 훈련 노드의 label rate
int(data.train_mask.sum())/data.num_nodes
## isolated node 존재 여부
data.has_isolated_nodes()
## self-loop 존재 여부
data.has_self_loops()
```

- Data
  - `print(data)` 가 보여주는 속성들
    - `edge index`

      graph connectivity - 각 edge에 대한 source와 destination node indices를 가진 tuple
    - `x`

      node features
    - `y`

      node labels - 각각의 node는 단 하나의 class에만 속해 있음
    - `train_mask`

      우리가 label을 알고 있는 node의 개수 (training node)
  - visualization

```
from torch_geometric.utils import to_networkx

G=to_networkx(data,to_undirected=True)
visualize(G,color=data.y) ## y에 따라 색깔 구분
```

- Implementing Graph Neural Networks (GNN)
  - GNN의 output
    - embedding

각 node가 feature vector를 갖고 있는 graph를 인풋으로 받는 function을 훈련시켜 downstream task에 사용할 수 있는 vector

- 이렇게 GNN으로 얻은 embedding vector을 node/edge/graph level regression이나 classification에 사용할 수 있음

  → 여기서는 각각의 node를 제대로 분류할 수 있는 embedding을 훈련시킴

- 모델 생성

```
import torch
from torch.nn import Linear
from torch_geometric.nn import GCNConv

class GCN(torch.nn.Module):
  def __init__(self):
      super(GCN,self).__init__()
      torch.manual_seed(12345)
      self.classifier=Linear(2,dataset.num_classes)

      self.convs=torch.append(GCNConv(input_dim,hidden_dim))
      for l in range(num_layers-1):
        self.convs.append(GCNConv(hidden_dim,hidden_dim))

      self.relu=torch.nn.ReLU()

    def forward(self,x,edge_index):
        for l in range(num_layers):
          x=self.convs[l](x,edge_index)
          x=x.tanh()

        h=torch.nn.functional.relu(h)
        h=torch.nn.functional.dropout(h,dropout=0.5,training=self.training)
        h=self.conv3(h,edge_index)
        embeddings=h.tanh()

        out=self.classifier(embeddings)

        return out,embeddings


## 다른 방법
class GCN(torch.nn.Module):
  def __init__(self):
      super().__init__()
      torch.manual_seed(12345)
      self.conv1=GCNConv(data.num_features,4)
      self.conv2=GCNConv(4,4)
      self.conv3=GCNConv(4,2)
      self.classifier=Linear(2,data.num_classes)

  def forward(self,x,edge_index):
      h=self.conv1(x,edge_index)
      h=h.tanh()
      h=self.conv2(h,edge_index)
      h=h.tanh()
      h=self.conv3(h,edge_index)
      h=h.tanh()

      out=self.classifier(h)

      return out,h

model=GCN()
print(model)
```

- 모델 훈련 및 평가

```
import time

model = GCN()
criterion = torch.nn.CrossEntropyLoss()  # Define loss criterion.
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)  # Define optimizer.

def train(data):
    optimizer.zero_grad()  # Clear gradients.
    out, h = model(data.x, data.edge_index)  # Perform a single forward pass.
```

```
    loss = criterion(out[data.train_mask], data.y[data.train_mask])  # Compute the loss solely based on the training nodes.
    loss.backward()  # Derive gradients.
    optimizer.step()  # Update parameters based on gradients.

    accuracy = {}
    # Calculate training accuracy on our four examples
    predicted_classes = torch.argmax(out[data.train_mask], axis=1) # [0.6, 0.2, 0.7, 0.1] -> 2
    target_classes = data.y[data.train_mask]
    accuracy['train'] = torch.mean(
        torch.where(predicted_classes == target_classes, 1, 0).float())

    # Calculate validation accuracy on the whole graph
    predicted_classes = torch.argmax(out, axis=1)
    target_classes = data.y
    accuracy['val'] = torch.mean(
        torch.where(predicted_classes == target_classes, 1, 0).float())

    return loss, h, accuracy

for epoch in range(500):
    loss, h, accuracy = train(data)
    # Visualize the node embeddings every 10 epochs
    if epoch % 10 == 0:
        visualize(h, color=data.y, epoch=epoch, loss=loss, accuracy=accuracy)
        time.sleep(0.3)
```