Code Review

(GraphSAGE) Link prediction Tutorial

배홍섭, 손재성, 송우진

INDEX

Introduction

Introduction

❑ 목적 : 링크 예측(link prediction)

- 그래프에서 임의의 두 노드 사이에 에지의 존재를 예측하기 위해 GNN을 train 시켜보는 예제

❑ Background

- 소셜 추천, 아이템 추천, 지식 그래프 완성 등과 같은 많은 애플리케이션은 두 특정 노드 사이에 에지가 존재하는지 여부를 예측하는 링크 예측을 통해 수행될 수 있다.

❑ Overview

1. GNN 기반 링크 예측 모델을 구축.
2. DGL 제공 데이터 세트에서 모델을 train, test

- # Setting

Setting

DGL 라이브러리 설치

```
[ ]  !pip install dgl
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: dgl in /usr/local/lib/python3.8/dist-packages (0.9.1)
Requirement already satisfied: networkx>=2.1 in /usr/local/lib/python3.8/dist-packages (from dgl) (2.8.8)
Requirement already satisfied: numpy>=1.14.0 in /usr/local/lib/python3.8/dist-packages (from dgl) (1.21.6)
Requirement already satisfied: tqdm in /usr/local/lib/python3.8/dist-packages (from dgl) (4.64.1)
Requirement already satisfied: requests>=2.19.0 in /usr/local/lib/python3.8/dist-packages (from dgl) (2.25.1)
Requirement already satisfied: scipy>=1.1.0 in /usr/local/lib/python3.8/dist-packages (from dgl) (1.7.3)
Requirement already satisfied: psutil>=5.8.0 in /usr/local/lib/python3.8/dist-packages (from dgl) (5.9.4)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in /usr/local/lib/python3.8/dist-packages (from requests>=2.19.0->dgl) (1.24.3)
Requirement already satisfied: chardet<5,>=3.0.2 in /usr/local/lib/python3.8/dist-packages (from requests>=2.19.0->dgl) (4.0.0)
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.8/dist-packages (from requests>=2.19.0->dgl) (2.10)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.8/dist-packages (from requests>=2.19.0->dgl) (2022.12.7)
```

필요 라이브러리 로드

```
[ ]  # 필요 라이브러리 Load
     import dgl
     import torch
     import torch.nn as nn
     import torch.nn.functional as F
     import itertools
     import numpy as np
     import scipy.sparse as sp
```

Setting

Graph and features 로드

Cora dataset 로드한다.

```
[ ]  import dgl.data

     dataset = dgl.data.CoraGraphDataset()
     g = dataset[0]
```

```
  NumNodes: 2708
  NumEdges: 10556
  NumFeats: 1433
  NumClasses: 7
  NumTrainingSamples: 140
  NumValidationSamples: 500
  NumTestSamples: 1000
Done loading data from cached files.
```

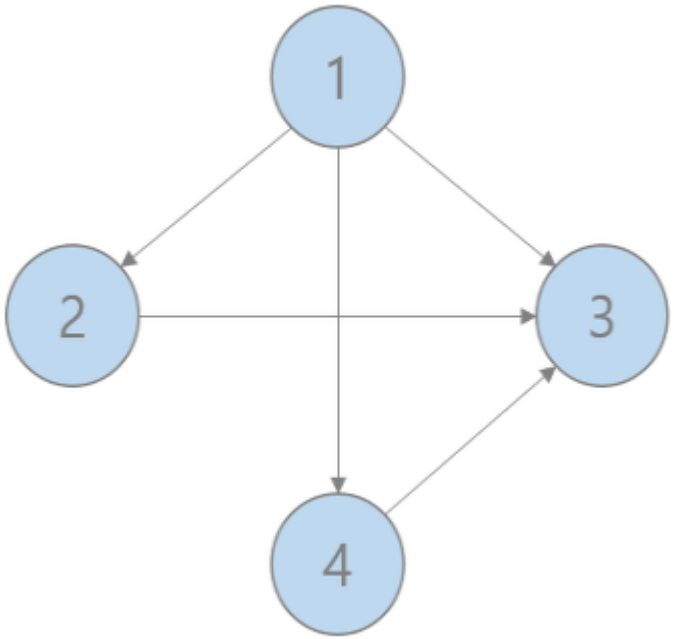
g : dgl 라이브러리에서 가져온 CoraGraphDataset

즉 g는 그래프 데이터.

- # Define train set and test set

Define train set and test set

```
u, v = g.edges() # u : 시작 노드 , v : 도착 노드 (Tensor type)
```



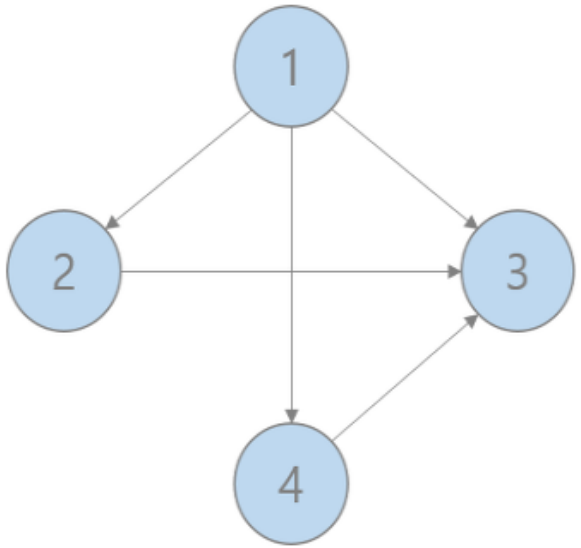
- Edge(=link)는 시작점과 끝점으로 표현될 수 있다.

  1에서 2로 가는 간선(edge)는 시작점:1, 도착점:2로 표현될 수 있는데 이를 u=1, v=2로 표현함으로써 해당 간선을 표현할 수 있다.

- 따라서 u는 시작노드, v는 도착노드 들의 집합이다.

Define train set and test set

```
eids = np.arange(g.number_of_edges()) # g.number_of_edges() : int type -> eids : ndarray type
# np.arange(g.number_of_edges()) : array([0, 1, 2, 3, 4])
eids = np.random.permutation(eids) # shuffle
# eids = array([3, 4, 1, 0, 2])
```



g.number_of_edges() : 그래프 내의 간선 개수를 return

- **Ex) return 5 (오른쪽 그래프에서 edge는 5개)**

np.arange(n) : 0부터n-1까지 value를 가지는 ndarray return

- **즉 np.arange(g.number_of_edges()) 는 그래프(g) 내의 간선 개수 크기의 ndarray 생성**
- **Ex) array([0,1,2,3,4])**

np.random.permutation(list) : list의 원소 위치를 섞고 return(shuffle)

- **Ex) array([3,4,1,0,2])**

Brief conclusion:

- eids는 edge에 id를 할당

Define train set and test set

Graph and features 로드

Cora dataset 로드한다.

```
[ ]  import dgl.data

     dataset = dgl.data.CoraGraphDataset()
     g = dataset[0]
```

```
  NumNodes: 2708
  NumEdges: 10556
  NumFeats: 1433
  NumClasses: 7
  NumTrainingSamples: 140
  NumValidationSamples: 500
  NumTestSamples: 1000
Done loading data from cached files.
```

```
test_size = int(len(eids) * 0.1) # 1055개
train_size = g.number_of_edges() - test_size # 10556 - 1055
```

앞서 내용을 통해 eids 는 random shuffle 된 ndarray임을 알 수 있다.

그 전에 소개했던 dataset에 대해 다시 살펴보자면 edge는 총 10,556 개 임을 확인할 수 있다.

test set은 전체 dataset 중 10%만 사용하는 것을 확인할 수 있다.

또한 train set는 전체 dataset에서 test set을 뺀 나머지이다.

Define train set and test set

```
# u, v에서 같은 index(i)는 시작 노드(u[i])와 도착 노드(v[i]) 사이의 연결된 edge를 의미한다
test_pos_u, test_pos_v = u[eids[:test_size]], v[eids[:test_size]] # test size 만큼 pair of nodes(=edges) 형태로 저장한다
train_pos_u, train_pos_v = u[eids[test_size:]], v[eids[test_size:]] # train size 만큼 pair of nodes(=edges) 형태로 저장한다
```

u,v 는 각각 시작노드, 도착노드 들의 집합.

eids 는 전체 edge에 대해 id를 가지고 있는 ndarray

따라서 eids(=ndarray) 에서 test size 개수 만큼 test_pos_u, test_pos_v 로 정의
같은 방법으로 train_pos_u, train_pos_v 정의

pos 의미?

- **그래프를 표현할 때 node 사이에 edge가 존재할 수도, 존재하지 않을 수도 있다.**
- **이를 학습하여 input dataset에 대해 link의 여부를 학습하는 것으로,**
  - **node 사이에 edge가 존재하면 positive examples로,**
  - **node 사이에 edge가 존재하지 않는다면 negative examples로 분류**

Define train set and test set

```
# Find all negative edges and split them for training and testing
adj = sp.coo_matrix((np.ones(len(u)), (u.numpy(), v.numpy()))) # coo_matrix로 인접 행렬 생성한다
# sp.coo_matrix((data, (row, col))) , coo_matrix Reference : https://radish-greens.tistory.com/1
adj_neg = 1 - adj.todense() - np.eye(g.number_of_nodes())
# adj.todense() : sp matrix(정보 압축 상태) -> numpy matrix로 형변환 해준다.
# np.eye() : identity matrix (단위 행렬)
# adj_neg = edge가 존재하지 않으면 1, edge가 존재하면 0 (self-loop는 고려 X -> 0으로 만든다)
neg_u, neg_v = np.where(adj_neg != 0)
# edge가 존재하지 않은 값들에 대해서 negative examples 생성
```

필요 라이브러리 로드

```
[ ]  # 필요 라이브러리 Load
     import dgl
     import torch
     import torch.nn as nn
     import torch.nn.functional as F
     import itertools
     import numpy as np
     import scipy.sparse as sp
```

sp.coo_matrix : spicy 라이브러리에서 제공하는 function으로 COO 형식으로 matrix 생성

- **COO 형식이란 COOrdinate 줄임말로 좌표로 행렬을 표현하는 방식을 말한다. (다음 슬라이드에서 설명)**

Define train set and test set

- **COO 형식이란 ?**
  - COOrdinate 줄임말로 좌표로 행렬을 표현하는 방식을 말한다.

- **COO matrix : 행렬의 희소 표현을 위해 사용**
  - 행렬의 대부분의 값이 0 이라면 모든 값을 저장하는 것은 메모리 낭비이므로.
  - 값이 존재하는 칸의 행과 열 그리고 값만 저장하여 적은 메모리로 동일한 행렬을 표현할 수 있다.

2	4	2	0
0	0	1	0
0	0	0	5

기존 matrix 표현

행 0, 열 0 에 value 2	행 1, 열 0 에 value 0	행 2, 열 0 에 value 0
행 0, 열 1 에 value 4	행 1, 열 1 에 value 0	행 2, 열 1 에 value 0
행 0, 열 2 에 value 2	행 1, 열 2 에 value 1	행 2, 열 2 에 value 0
행 0, 열 3 에 value 0	행 1, 열 3 에 value 0	행 2, 열 3 에 value 5

COO matrix 표현
값이 존재하는 칸 : 2, 4, 2, 1, 5
값이 존재하는 행 : 0, 0, 0, 1, 2
값이 존재하는 열 : 0, 1, 2, 2, 3

행렬에 0인 값들이 많은 희소 행렬이라면, 그 정도가 심할 수록 메모리 오버헤드 방지를 위해 COO 형식을 사용하여 COO matrix로 표현해야 한다.

Define train set and test set

```
# Find all negative edges and split them for training and testing
adj = sp.coo_matrix((np.ones(len(u)), (u.numpy(), v.numpy()))) # coo_matrix로 인접 행렬 생성한다
# sp.coo_matrix((data, (row, col))) , coo_matrix Reference : https://radish-greens.tistory.com/1
adj_neg = 1 - adj.todense() - np.eye(g.number_of_nodes())
# adj.todense() : sp matrix(정보 압축 상태) -> numpy matrix로 형변환 해준다.
# np.eye() : identity matrix (단위 행렬)
# adj_neg = edge가 존재하지 않으면 1, edge가 존재하면 0 (self-loop는 고려 X -> 0으로 만든다)
neg_u, neg_v = np.where(adj_neg != 0)
# edge가 존재하지 않은 값들에 대해서 negative examples 생성
```

필요 라이브러리 로드

```
[ ]  # 필요 라이브러리 Load
     import dgl
     import torch
     import torch.nn as nn
     import torch.nn.functional as F
     import itertools
     import numpy as np
     import scipy.sparse as sp
```

sp.coo_matrix : spicy 라이브러리에서 제공하는 function으로 COO 형식으로 matrix 생성

- 따라서 adj(인접행렬) 은 coo 형식으로 표현된 matrix
- adj에는 edge가 존재할 경우 1 을 가지는 matrix의 row, col 정보가 들어있다.

adj.todense() : matrix 연산을 하기 위해서는 COO 형식(압축 형태)로는 불가능 하기 때문에 일반 matrix형태로 복원시켜야 함.

np.eye(n) : np.identity()와 동일한 function으로 크기가 n인 항등행렬 생성

adj_neg : 앞서 negative examples 정의에 대해 보았듯이 node 사이에 edge가 없는 경우에 해당한다.

- 따라서 adj 이외의 값에 대해서 1을 부여하고, self-loop 는 해당 튜토리얼에서 고려하지 않기 때문에 제외.
- adj_neg = 1 – adj.todense() – np.eye(g.number_of_nodes())

neg_u, neg_v : adj_neg 에서 0이 아닌, 즉 1인 값에 대해 시작노드, 도착노드 들의 집합을 의미.

Define train set and test set

```
neg_eids = np.random.choice(len(neg_u), g.number_of_edges())
# len(neg_u) 즉, negative examples는 7320000가 존재하지만 positive examples 만큼 추출하여 sample 수를 맞춰준다.

test_neg_u, test_neg_v = neg_u[neg_eids[:test_size]], neg_v[neg_eids[:test_size]] # test size 만큼 pair of nodes(=edges) 형태로 저장한다
train_neg_u, train_neg_v = neg_u[neg_eids[test_size:]], neg_v[neg_eids[test_size:]] # train size 만큼 pair of nodes(=edges) 형태로 저장한다
```

np.random.choice(num, count) : num 이내 숫자에 대해 count 개수 만큼 랜덤 뽑기

- **count = g.number_of_edges() 이기 때문에 그래프(g)에 존재하는 edge수 만큼 뽑게 된다.**
- **Node 가 2708 임을 고려하면 전체 가능한 edge 수는 2708 * 2708, 그 중 10,556개가 존재.**

test_pos_u, test_pos_v 를 정의하는 방식과 동일하게 test_neg_u, test_neg_v 정의.

train_neg_u, train_neg_v 또한 동일한 방식으로 정의.

Graph and features 로드

Cora dataset 로드한다.

```
[ ]  import dgl.data

     dataset = dgl.data.CoraGraphDataset()
     g = dataset[0]
```

```
  NumNodes: 2708
  NumEdges: 10556
  NumFeats: 1433
  NumClasses: 7
  NumTrainingSamples: 140
  NumValidationSamples: 500
  NumTestSamples: 1000
Done loading data from cached files.
```

Define train set and test set

학습할 때 원본 그래프에서 test set의 edge를 제거(=train set) 해야 한다.

```
dgl.remove_edges
```

```
[ ]  train_g = dgl.remove_edges(g, eids[:test_size])
     # Train Gragh : num_nodes=2708, num_edges=9501
```

최종적으로 train set, 즉 input으로 들어갈 train graph를 정의.

이때 train graph에서는 test 시 사용할 edge에 대해서는 제거 해주어야 한다.

- **Input으로 들어가는 graph에 test 시 예측 정확도를 비교하기 위한 edge가 존재할 경우, 이에 대해서도 학습을 진행하기 때문에**

Breif conclusion:

- **Input으로 사용될 train_g를 정의하였다.**
- **Train 과 test시 사용될 u,v(=edge)에 대해서도 정의하였다.**
- **특히 positive examples, negative examples에 대해서 정의하였다.**

- # Model 정의

Model 정의

```
from dgl.nn import SAGEConv
# dgl.nn : Pytorch nn과 동일한 라이브러리

# ----------- 2. create model -------------- #
# build a two-layer GraphSAGE model
class GraphSAGE(nn.Module):
    def __init__(self, in_feats, h_feats):
        super(GraphSAGE, self).__init__()
        self.conv1 = SAGEConv(in_feats, h_feats, 'mean')
        self.conv2 = SAGEConv(h_feats, h_feats, 'mean')

    def forward(self, g, in_feat):
        h = self.conv1(g, in_feat)
        h = F.relu(h)
        h = self.conv2(g, h)
        return h
```

```
# aggregator function 3가지(mean, pool, LSTM) 중 하나를 적용한다.
```

- GraphSAGE 모델 정의 클래스 (__init__)

예제에 사용되는 GraphSAGE 모델은 다음과 같이 2개의 SAGEConv를 정의하고 있다.

- GraphSAGE 모델 정의 클래스 (forward)

(g = Graph, in_feat = Feature)

데이터 g와 in_feat을 입력 받아 SAGEConv1에 넣어주고 activation function에 넣어주는 과정을 거치며 이후 SAGEConv2에 넣어주는 과정을 거친다.

- 활용

Aggregator function 변경 가능 - (mean, pool, LSTM) 중 하나 적용 가능하다

Model 정의

```
# Graph : dgl.graph((u, v), num_nodes = g.number_of_nodes()) 만들어 준다.
# (u, v) : edge의 정보
# num_nodes = g.number_of_nodes() : 노드의 개수

# Graph 표현은 Edge List로 표현된다.
train_pos_g = dgl.graph((train_pos_u, train_pos_v), num_nodes=g.number_of_nodes())
train_neg_g = dgl.graph((train_neg_u, train_neg_v), num_nodes=g.number_of_nodes())

test_pos_g = dgl.graph((test_pos_u, test_pos_v), num_nodes=g.number_of_nodes())
test_neg_g = dgl.graph((test_neg_u, test_neg_v), num_nodes=g.number_of_nodes())
```

- ***positive graph & negative graph***

link prediction을 하기 위해서는 pair of nodes(=edge)를 계산해야 한다.

DGL 라이브러리를 사용하여 *positive graph* 와 *negative graph*를 구성할 수 있다.

- **Graph 구성**

Train data 와 Test data에 *positive graph* 와 *negative graph*를 구성하였다.

Graph 표현은 Edge List로 표현된다. ex) `[(0, 1), (1, 2), (2, 3), (0, 2), (3, 2), (4, 5), (5, 4)]`

- GraphSAGE에서 나오는 weight 값들에 대해서 마지막에 결과 예측을 하기 위해 아래의 클래스를 사용한다.
- 계산방법에 따라 Tutorial에서는 두가지로 계산할 수 있다.

DotPredictor()는 element-wise dot를 활용하여 message passing 방법으로 'score'를 계산한다.

MLPPredictor()는 MLP(Multi-layer Perceptron)을 활용하여 message passing 방법으로 'score'를 계산한다.

```
import dgl.function as fn

# GraphSAGE에서 나오는 weight 값들에 대해서 마지막에 결과 예측 하기 위해 사용한다.
class DotPredictor(nn.Module):
    def forward(self, g, h):
        with g.local_scope():
            g.ndata['h'] = h  # dictionary에 ('h' : key , h : value) 추가
            g.apply_edges(fn.u_dot_v('h', 'h', 'score')) # apply_edges() : new edge features를 기존의 edge features 와 node features에 기반하여 계산해 주는 함수이다.
            # 추가 Reference : https://docs.dgl.ai/en/0.2.x/generated/dgl.DGLGraph.apply_edges.html 참조

            return g.edata['score'][:, 0] # .apply_edges를 이용해서 'score'를 계산하고 리턴해준다.
```

Model 정의

```
import dgl.function as fn

# GraphSAGE에서 나오는 weight 값들에 대해서 마지막에 결과 예측 하기 위해 사용한다.
class DotPredictor(nn.Module):
    def forward(self, g, h):
        with g.local_scope():
            g.ndata['h'] = h  # dictionary에 ('h' : key , h : value) 추가
            g.apply_edges(fn.u_dot_v('h', 'h', 'score'))
            # 추가 Reference : https://docs.dgl.ai/en/0.2.x/generated/dgl.DGLGraph.apply_edges.

            return g.edata['score'][:, 0] # .apply_edges를 이용해서 'score'를 계산하고 리턴해준다.
```

- DotPredictor()는 element-wise dot를 활용하여 message passing 방법으로 'score'를 계산한다.

- fn.u_dot_v(feature field of u, feature field of v, output message field) : message passing 방법

- apply_edges() : new edge features를 기존의 edge features와 node features에 기반하여 계산해 주는 함수

- .apply_edges()를 이용해서 ‘score’를 계산하고 return 해준다

Model 정의

```python
# 위 과정이 이해하기 어려운 경우 자체적으로 아래의 함수를 이용해 사용 가능하다
class MLPPredictor(nn.Module):
    def __init__(self, h_feats):
        super().__init__()
        self.W1 = nn.Linear(h_feats * 2, h_feats)
        self.W2 = nn.Linear(h_feats, 1)

    def apply_edges(self, edges):
        """
        Computes a scalar score for each edge of the given graph.

        Parameters
        ----------
        edges :
            Has three members ``src``, ``dst`` and ``data``, each of
            which is a dictionary representing the features of the
            source nodes, the destination nodes, and the edges
            themselves.

        Returns
        -------
        dict
            A dictionary of new edge features.
        """
        h = torch.cat([edges.src['h'], edges.dst['h']], 1)
        return {'score': self.W2(F.relu(self.W1(h))).squeeze(1)}

    def forward(self, g, h):
        with g.local_scope():
            g.ndata['h'] = h
            g.apply_edges(self.apply_edges)
            return g.edata['score']
```

- MLPPredictor()는 MLP(Multi-layer Perceptron)을 활용하여 message passing 방법으로 'score'를 계산한다.

- # Model Train

Model Train

- **전체 모델, 손실함수(loss function), 평가 지표 정의**

1. 전체 모델은 위에서 Model을 정의한 것과 같다.
2. 손실 함수(loss function)는 simply binary cross entropy loss를 활용한다.

$$\mathcal{L} = -\sum_{u \sim v \in \mathcal{D}} (y_{u \sim v} \log(\hat{y}_{u \sim v}) + (1 - y_{u \sim v}) \log(1 - \hat{y}_{u \sim v})))$$

3. 평가지표는 AUC(Area Under the ROC Curve)를 활용한다.

Model Train

```
model = GraphSAGE(train_g.ndata['feat'].shape[1], 16)
# 이 Tutorial에서는 DotPredictor()를 사용하지만 MLPPredictor()로도 대체 가능하다.
# pred = MLPPredictor(16)
pred = DotPredictor()
```

- **Train 시작**

train_g.ndata(=Train Graph의 Node data)는 다음과 같이 구성되어 있다.

```
{'train_mask': tensor([ True,  True,  True,  ..., False, False, False]), 'val_mask': tensor([False, False, False,  ..., False, False, False]), 'test_mask':
tensor([False, False, False,  ...,  True,  True,  True]), 'label': tensor([3, 4, 4,  ..., 3, 3, 3]), 'feat': 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.]])}
```

이 중 ['feat'] 데이터를 사용한다 → [feature data = 1433]

pred는 DotPredictor()를 사용한다. (MLPPredictor도 사용 가능하다)

Model Train

```
# score(예측)값를 가지고 label(실제)값과 비교하여 loss와 정확도를 계산한다.
def compute_loss(pos_score, neg_score):
    scores = torch.cat([pos_score, neg_score])
    labels = torch.cat([torch.ones(pos_score.shape[0]), torch.zeros(neg_score.shape[0])])
    return F.binary_cross_entropy_with_logits(scores, labels)

def compute_auc(pos_score, neg_score):
    scores = torch.cat([pos_score, neg_score]).numpy() # 예측 값
    labels = torch.cat(
        [torch.ones(pos_score.shape[0]), torch.zeros(neg_score.shape[0])]).numpy() # 실제 값
    return roc_auc_score(labels, scores)
```

- Score(=예측) 값을 가지고 label(=실제) 값과 비교하여 Loss와 정확도를 계산한다

- Loss function은 binary cross entropy loss를 활용한다.
- 평가지표는 roc_auc_score를 활용한다.

Optimizer

```
# ----------- 3. set up loss and optimizer -------------- #
# in this case, loss will in training loop
optimizer = torch.optim.Adam(itertools.chain(model.parameters(), pred.parameters()), lr=0.01)
```

- Optimizer 정의

일반적인 딥러닝에서 쓰이는 Adam을 사용하여 최적화를 진행한다.

효율적인 concat을 위해 itertools.chain을 사용한다

Model Train

```
# ----------- 4. training ------------------------------- #
for e in range(100):
    # forward
    h = model(train_g, train_g.ndata['feat'])
    pos_score = pred(train_pos_g, h)
    neg_score = pred(train_neg_g, h)
    loss = compute_loss(pos_score, neg_score)

    # backward
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if e % 5 == 0: # epoch 5번 마다 출력한다.
        print('In epoch {}, loss: {}'.format(e, loss))
```

- Forward

model(=GraphSAGE) 객체에 train_g(그래프)와

train_g.ndata['feat']를 전달 시 h를 반환

positive score 와 *negative score*를 비교하여 loss값을 계산

- Backward

optimizer.zero_grad() : 파라미터 초기화

loss.backward() : loss값을 기준으로 gradient 계산

optimizer.step() : loss값을 기준으로 계산한 gradient를

optimizer를 사용해 파라미터 update를 진행 (backpropagation)

- # Model Evaluate

Model Train

- **Train 실행 결과**

```
In epoch 0, loss: 0.6929768919944763
In epoch 5, loss: 0.6621994972229004
In epoch 10, loss: 0.5646780133247375
In epoch 15, loss: 0.511516809463501
In epoch 20, loss: 0.4748401939868927
In epoch 25, loss: 0.4437951445579529
In epoch 30, loss: 0.42105838656425476
In epoch 35, loss: 0.39862629771232605
In epoch 40, loss: 0.37493836879730225
In epoch 45, loss: 0.35579973459243774
In epoch 50, loss: 0.33541208505630493
In epoch 55, loss: 0.3151055872440338
In epoch 60, loss: 0.2949903905391693
In epoch 65, loss: 0.274679034948349
In epoch 70, loss: 0.25403541326522827
In epoch 75, loss: 0.23329827189445496
In epoch 80, loss: 0.2128126323223114
In epoch 85, loss: 0.19286388158798218
In epoch 90, loss: 0.17312371730804443
In epoch 95, loss: 0.15436574816703796
```

Model Train

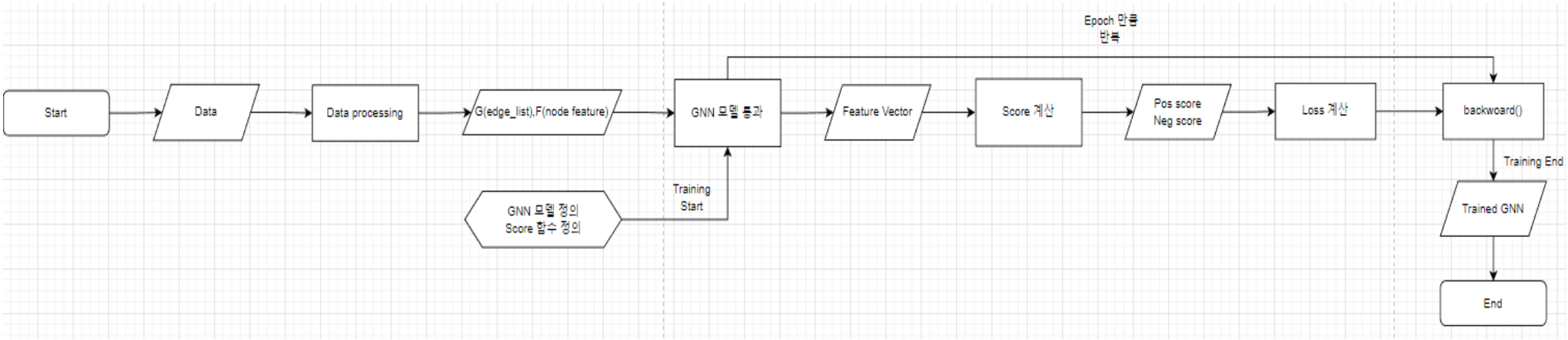
```
# ----------- 5. check results ----------------------- #
from sklearn.metrics import roc_auc_score
with torch.no_grad():
    pos_score = pred(test_pos_g, h)
    neg_score = pred(test_neg_g, h)
    print('AUC', compute_auc(pos_score, neg_score))
```

- 성능평가 진행 AUC 0.8512791716268727

positive score 와 *negative score*를 비교하여 AUC값을 계산

결과는 AUC가 0.851 정도 나온다

Flow Chart


Epoch 만큼
반복
Start
Data
Data processing
G(edge_list),F(node feature)
GNN 모델 통과
Feature Vector
Score 계산
Pos score
Neg score
Loss 계산
backwoard()
Training End
Trained GNN
End
GNN 모델 정의
Score 함수 정의
Training
Start

Node Classification vs Link Prediction

	Node Classification	Link Prediction
모델 인풋	G(Edge_list),F(Node Feature)	A(인접행렬),F(Node Feature)
모델 아웃풋	Classification result	M by N Feature Vector
Score Func	X	O
차이점	모델에 인풋데이터를 넣으면 바로 결과 도출	모델에서 나온 Feature Vector를 통해서 Score를 계산하고 Pos,Neg로 구분하여 Link가 있을 것인지 예측함