MLG:
Machine Learning with Graphs (*Strojno učenje na grafih*)

Assignment: Final Project

Submission time: 02:30 **and date:** 22. 03. 2023

Submission Fill in and include this cover sheet with each of your assignments. It is an honor code violation to write down the wrong date and/or time. Assignments are due at 9:00am and should be submitted through Gradescope and eUcilnica. Students should check Piazza for submission details.

Late Periods Each student will have a total of *two* free late periods. *One late period expires the morning on the day before the next class.* (Assignments are usually due on Fridays, which means the first late period expires on the following Tuesday at 9:00am.) Once these late periods are exhausted, any assignments turned in late will be penalized 50% per late period. However, no assignment will be accepted more than *one* late period after its due date.

Honor Code We strongly encourage students to form study groups. Students may discuss and work on assignments in groups. However, each student must write down their solutions independently, i.e., each student must understand the solution well enough in order to reconstruct it by him/herself. Students should clearly mention the names of all the other students who were part of their study group. Using code or solutions obtained from the web (GitHub, Google, previous years etc.) is considered an honor code violation. We check all submissions for plagiarism. We take the honor code very seriously and expect students to do the same.

Name: Yon Ploj

Email: yp8702@student.uni-lj.si **SID:** 63200025

Study group: Team Kul (Aleksander Piciga, Jakob Drusany, Gregor Alič, Yon Ploj)

I acknowledge and accept the Honor Code.

(Signed) [illegible]

Medium article:

https://medium.com/@drusany.jakob/playlist-expansion-with-

Github repository:

https://github.com/plojyon/mlg

In this project, we will train a GNN to perform link prediction on a heterogenous graph from the Spotify Million Playlists dataset.

Import libraries

```
!pip install torch numpy matplotlib torch_geometric torch-scatter torcheval
# https://stackoverflow.com/a/73534928
import torch
!pip install torch-sparse -f https://data.pyg.org/whl/torch-{torch.__version__}.html
```

```
import tqdm
import torch
import torch_geometric
import numpy as np
import time
import matplotlib.pyplot as plt
import torch_geometric.transforms as T
from torcheval.metrics import BinaryAccuracy

import itertools
import time
import networkx as nx
import json
import os

import networkx as nx
from tqdm import tqdm
```

Configuration

Dataset files

```
from google.colab import drive
drive.mount('/content/drive')

dataset_path = "drive/MyDrive/spotify_million_playlist_dataset/data"
```

```
    Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

Load graph

```
def load_graph(dataset_path=dataset_path):
    """Load a nx.Graph from disk."""
    filenames = os.listdir(dataset_path)
    G = nx.DiGraph()
    for i in tqdm(range(len(filenames)), unit="files"):
        with open(os.path.join(dataset_path, filenames[i])) as json_file:
            playlists = json.load(json_file)["playlists"]
            for playlist in playlists:
                playlist_name = f"spotify:playlist:{playlist['pid']}"
                G.add_node(
                    playlist_name,
                    node_type="playlist",
                    num_followers=playlist["num_followers"],
                    num_tracks=playlist["num_tracks"],
                    num_artists=playlist["num_artists"],
                    num_albums=playlist["num_albums"],
                    duration_ms=playlist["duration_ms"],
                    collaborative=playlist["collaborative"],
                    num_edits=playlist["num_edits"]
                )
                for track in playlist["tracks"]:
                    G.add_node(track["track_uri"], node_type="track", duration=track["duration_ms"])
                    G.add_node(track["album_uri"], node_type="album")
                    G.add_node(track["artist_uri"], node_type="artist")

                    G.add_edge(track["track_uri"], playlist_name, edge_type="track-playlist")
                    G.add_edge(track["track_uri"], track["album_uri"], edge_type="track-album")
                    G.add_edge(track["track_uri"], track["artist_uri"], edge_type="track-artist")
    return G

def nx2hetero(G):
```

```
"""Convert a nx.Graph into a torch_geometric.data.HeteroData object."""
ids_by_type = {
    "playlist": {},
    "track": {},
    "artist": {},
    "album": {}
}

def node_id(node_type, id):
    d = ids_by_type[node_type]
    if id not in d:
        d[id] = len(d)
    return d[id]

node_features_by_type = {
    "playlist": [],
    "track": [],
    "artist": [],
    "album": []
}

# comment:
# {
#     "name": "musical",
#     "collaborative": "false",
#     "pid": 5,
#     "modified_at": 1493424000,
#     "num_albums": 7,
#     "num_tracks": 12,
#     "num_followers": 1,
#     "num_edits": 2,
#     "duration_ms": 2657366,
#     "num_artists": 6,
#     "tracks": [
#         {
#             "pos": 0,
#             "artist_name": "Degiheugi",
#             "track_uri": "spotify:track:7vqa3sDmtEaVJ2gcvxtRID",
#             "artist_uri": "spotify:artist:3V2paBXEoZIAhfZRJmo2jL",
#             "track_name": "Finalement",
#             "album_uri": "spotify:album:2KrRMJ9z7Xjoz1Az4O6UML",
#             "duration_ms": 166264,
#             "album_name": "Dancing Chords and Fireflies"
#         },
#     ],

# })

for node in G.nodes(data=True):
    t = node[1]["node_type"]
    node_id(t, node[0])
    if t == "playlist":
        if node[1]["collaborative"] not in ("true", "false"):
            raise ValueError(f"collaborative is not a boolean: {node[1]['collaborative']}")
        node_features_by_type["playlist"] += [[
            node[1]["num_followers"],
            node[1]["collaborative"] == 'true',
            node[1]["num_albums"],
            node[1]["num_tracks"],
            node[1]["num_edits"],
            node[1]["duration_ms"],
            node[1]["num_artists"]
        ]]
    elif t == "track":
        distances = nx.single_source_shortest_path_length(G, node[0], cutoff=2)
        node_features_by_type["track"] += [[node[1]["duration"], len(distances)]]
    elif t == "artist":
        distances = nx.single_source_shortest_path_length(G, node[0], cutoff=2)
        node_features_by_type["artist"] += [[len(distances)]]
    elif t == "album":
        distances = nx.single_source_shortest_path_length(G, node[0], cutoff=2)
        node_features_by_type["album"] += [[len(distances)]]

edge_index_by_type = {
    ("track", "contains", "playlist"): [],
    ("track", "includes", "album"): [],
    ("track", "authors", "artist"): []
}
existing_edges = set()
for edge in G.edges(data=True):
    track_node = edge[0]
    other_node = edge[1]
    if "track" not in track_node:
        track_node, other_node = other_node, track_node
        if (track_node, other_node) in existing_edges:
            continue

        if G[edge[0]][edge[1]]["edge_type"] == "track-playlist":
            s_id = node_id("track", track_node)
            d_id = node_id("playlist", other_node)

            edge_index_by_type[("track", "contains", "playlist")] += [(s_id, d_id)]

        elif G[edge[0]][edge[1]]["edge_type"] == "track-album":
            s_id = node_id("track", track_node)
            d_id = node_id("album", other_node)

            edge_index_by_type[("track", "includes", "album")] += [(s_id, d_id)]

        elif G[edge[0]][edge[1]]["edge_type"] == "track-artist":
            s_id = node_id("track", track_node)
            d_id = node_id("artist", other_node)

            edge_index_by_type[("track", "authors", "artist")] += [(s_id, d_id)]

        existing_edges.add((track_node, other_node))

    # construct HeteroData
    hetero = torch_geometric.data.HeteroData()

    # add initial node features
    hetero["playlist"].x = torch.FloatTensor(node_features_by_type["playlist"]).reshape(-1,len(node_features_by_type["playlist"][0]))
    hetero["track"].x = torch.FloatTensor(node_features_by_type["track"]).reshape(-1,len(node_features_by_type["track"][0]))
    hetero["artist"].x = torch.FloatTensor(node_features_by_type["artist"]).reshape(-1,len(node_features_by_type["artist"][0]))
    hetero["album"].x = torch.FloatTensor(node_features_by_type["album"]).reshape(-1,len(node_features_by_type["album"][0]))

    # add edge indices
    hetero["track", "contains", "playlist"].edge_index = torch.tensor(edge_index_by_type[("track", "contains", "playlist")]).t()
    hetero["track", "includes", "album"].edge_index = torch.tensor(edge_index_by_type[("track", "includes", "album")]).t()
    hetero["track", "authors", "artist"].edge_index = torch.tensor(edge_index_by_type[("track", "authors", "artist")]).t()

    # post-processing
    hetero = torch_geometric.transforms.ToUndirected()(hetero)
    hetero = torch_geometric.transforms.NormalizeFeatures()(hetero)
    return hetero

def ghetero2datasets(ghetero):
    """Split the dataset into train, validation and test sets."""
    transform = T.Compose([
        T.NormalizeFeatures(),
        T.RandomLinkSplit(
            num_val=0.1,
            num_test=0.1,
            disjoint_train_ratio=0.3,
            neg_sampling_ratio=2.0,
            add_negative_train_samples=False,
            edge_types=("track", "contains", "playlist"),
            rev_edge_types=("playlist", "rev_contains", "track"),
        )
    ])

    return transform(ghetero)  # 3-tuple: data_train, data_val, data_test
```

Preprocessing

We took a subset of our dataset by taking a subgraph of 5000 nodes.

```
def get_neigh_of_edge_type(G, edge_type, node):
    undirected_neigh = itertools.chain(G.neighbors(node), G.predecessors(node))
    return [
        n
        for n in undirected_neigh
        if (
            G.succ[node].get(n, dict()).get('edge_type', None) == edge_type or
            G.pred[node].get(n, dict()).get('edge_type', None) == edge_type
        )
    ]

def top_n_by_followers(G, n, node_type):
    """Get all nodes of type `node_type`."""
    playlists = [node for node in G.nodes(data=True) if node[1]["node_type"] == node_type]
    return sorted(playlists, key=lambda x:"num_followers" in x[1] and x[1]["num_followers"], reverse=True)[:n]
```

```
def get_smart_playlist_subset(G, playlists_to_keep):
    keep_nodes = set()
    for node in playlists_to_keep:
        keep_nodes.add(node[0])
        tracks = get_neigh_of_edge_type(G, "track-playlist", node[0])
        artists_and_albums = []

        for track in tracks:
            artists_and_albums += get_neigh_of_edge_type(G, "track-artist", track)
            artists_and_albums += get_neigh_of_edge_type(G, "track-album", track)

        keep_nodes = keep_nodes.union(set(tracks))
        keep_nodes = keep_nodes.union(set(artists_and_albums))
    return keep_nodes

def smart_split(G, splits=[100,500,1000,5000,10000]):
    ret = [None for _ in splits]
    for i in tqdm(splits):
        print(f"[{i}] started")
        start = time.time()
        playlists_to_keep = top_n_by_followers(G, i, "playlist")
        print(f"[{i}] got top n playlists in {time.time() - start} seconds")
        start = time.time()
        keep_nodes = get_smart_playlist_subset(G, playlists_to_keep)
        print(f"[{i}] finshed getting neighbors in {time.time() - start} seconds")
        print(f"\t({len(keep_nodes)} nodes = {len(keep_nodes)/len(G.nodes)} % of graph)")
        start = time.time()
        G_sub = nx.Graph(G.subgraph(keep_nodes))
        print(f"[{i}] finished subgraphing in {time.time() - start} seconds")
        start = time.time()
        splits[i] = G_sub
        print(f"[{i}] finished pickling in {time.time() - start} seconds")
    return ret
```

Model

Check if cuda is available

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

GNN embedding prediction network (three SAGEConv layers)

```
class GNN(torch.nn.Module):
    def __init__(self, hidden_channels):
        super().__init__()
        self.conv1 = torch_geometric.nn.SAGEConv((-1, -1), hidden_channels, normalize=True, dropout=True, bias=True, dropout_prob=0.1)
        self.conv2 = torch_geometric.nn.SAGEConv((-1, -1), hidden_channels, normalize=True, dropout=True, bias=True, dropout_prob=0.1)
        self.conv3 = torch_geometric.nn.SAGEConv((-1, -1), hidden_channels)

        self.reset_parameters()

    def forward(self, x, edge_index):
        x = self.conv1(x, edge_index)
        x = torch.nn.functional.leaky_relu(x, negative_slope=0.1)
        x = self.conv2(x, edge_index)
        x = torch.nn.functional.leaky_relu(x, negative_slope=0.1)
        x = self.conv3(x, edge_index)
        return x

    def reset_parameters(self):
        self.conv1.reset_parameters()
        self.conv2.reset_parameters()
        self.conv3.reset_parameters()
```

Link predictor (predicts using dot product)

```
class LinkPredictor(torch.nn.Module):

    def __init__(self):
        super().__init__()

    def forward(self, x_track, x_playlist, track_playlist_edge):
        track_embedding = x_track[track_playlist_edge[0]]
        playlist_embedding = x_playlist[track_playlist_edge[1]]

        # Apply dot-product to get a prediction per supervision edge:
        return (track_embedding * playlist_embedding).sum(dim=-1)
```

Full model

```
class HeteroModel(torch.nn.Module):
    def __init__(self, hidden_channels, node_features, metadata):
        super().__init__()
        self.node_lin = {
            k: torch.nn.Linear(v.shape[1], hidden_channels).to(device) for k, v in node_features.items()
        }
        self.gnn = GNN(hidden_channels).to(device)
        self.gnn = torch_geometric.nn.to_hetero(self.gnn, metadata=metadata).to(device)
        self.classifier = LinkPredictor().to(device)

    def embed(self, data):
        x_dict = {
            k: self.node_lin[k](v) for k, v in data.x_dict.items()
        }
        x_dict = self.gnn(x_dict, data.edge_index_dict)
        return x_dict

    def forward(self, data):
        x_dict = self.embed(data)
        pred = self.classifier(
            x_dict["track"],
            x_dict["playlist"],
            data["track", "contains", "playlist"].edge_label_index,
        )
        return pred

    def reset_parameters(self):
        for _, v in self.node_lin.items():
            torch.nn.init.xavier_uniform_(v.weight)
        self.gnn.reset_parameters()

def dummy_generator(source):
    for e in source:
        yield e
```

Model train and test functions

```
outs = []

def test(model, data_test):
    with torch.no_grad():
        test_out = model(data_test.to(device)).to('cpu')
        truth = data_test["track", "contains", "playlist"].edge_label.to('cpu')

    test_loss = torch.nn.functional.mse_loss(
        test_out,
        truth
    )
    metric = BinaryAccuracy()
    metric.update(test_out, truth)
    return float(test_loss), metric.compute()

def train(model, train_loader, optimizer, batch_wrapper=dummy_generator):
    model.train()

    accuracy = 0

    total_examples = total_loss = 0
    for batch in batch_wrapper(train_loader):
        optimizer.zero_grad()

        out = model(batch)
        truth = batch["track", "contains", "playlist"].edge_label

        loss = torch.nn.functional.mse_loss(
            out, truth
        )
        loss.backward()
        optimizer.step()

        aute_gledam = out.to('cpu')

        outs.append(aute_gledam)

        metric = BinaryAccuracy()
        metric.update(aute_gledam, truth.to('cpu'))
        accuracy += metric.compute() * len(out)
        total_examples += len(out)
        total_loss += float(loss) * len(out)

    return total_loss / total_examples, accuracy / total_examples
```

Train the model

```
G = load_graph()
ghetero = nx2hetero(G)
data_train, data_val, data_test = ghetero2datasets(ghetero)


# create training mask for playlist nodes
train_mask = torch.zeros(ghetero["playlist"].x.shape[0], dtype=torch.bool)
train_mask[torch.randperm(train_mask.shape[0])[:int(train_mask.shape[0]*0.8)]] = True

ghetero["playlist"].train_mask = train_mask

ghetero["playlist"].y = torch.LongTensor([1]*ghetero["playlist"].x.shape[0]).to(device)

model = HeteroModel(64, ghetero.x_dict, ghetero.metadata()).to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.0001, weight_decay=0.00001)
edge_label_index = data_train["track", "contains", "playlist"].edge_label_index
edge_label = data_train["track", "contains", "playlist"].edge_label

train_loader = torch_geometric.loader.LinkNeighborLoader(
    data=data_train,
    num_neighbors=[-1],
    neg_sampling_ratio=0.5,
    edge_label_index=(("track", "contains", "playlist"), edge_label_index),
    edge_label=edge_label,
    batch_size=20000,
    shuffle=True,
    transform=T.ToDevice(device)
)

epoch = 2000
render_graph = True

losses = []
accuracies = []
test_losses = []
test_accuracies = []

epoch_iter = tqdm(range(epoch), unit='epoch', desc='Training', bar_format='{desc:<5.5}{percentage:3.0f}%|{bar:10}{r_bar}')
for i in epoch_iter:
    loss, accuracy = train(model, train_loader, optimizer)
    losses.append(loss)
    accuracies.append(accuracy)
    test_loss, test_acc = test(model, data_val)
    test_losses.append(test_loss)
    test_accuracies.append(test_acc)
    epoch_iter.set_postfix_str(f"Train Loss: {loss:.4f}, Train Accuracy {accuracy:.4f}, Valid Loss {test_loss:.4f}, Valid Accuracy {test_acc:.4f}")
```

Render learning graph

```
plt.clf()
# add labels
plt.plot(np.arange(len(accuracies)), accuracies, label='Accuracy')
plt.plot(np.arange(len(losses)), losses, label='Loss')
plt.plot(np.arange(len(test_losses)), test_losses, label='Test Loss')
plt.plot(np.arange(len(test_accuracies)), test_accuracies, label='Test Accuracy')

# start plot at 0
plt.ylim(0, 1)
plt.legend()
plt.show()
```

1m 45s completed at 2:17 AM