Anomaly Detection
Lab 6 - Graph-based anomaly detection algorithms

Andrei Hîji

January 2025

In this lab we will use algorithms capable of identifying different types of anomalies in graphs. You will have to install the following libraries:

- **pip install networkx**
- **pip install torch torch-geometric**

1 Graph anomaly detection main ideas

- A type of graph convolutional operator that we will use in this lab is **GCNConv** from **from torch_geometric.nn**.

  $$\boldsymbol{X}^{\top} = \hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}X\Theta$$

  where $\hat{A} = A + I$ is the adjacency matrix with inserted self-loops and $\hat{D}_{i,i} = \Sigma_{j=0}\hat{A}_{i,j}$ is the diagonal degree matrix

A few methods that will be useful to you throughout this lab session are:

- **numpy.linalg.eigh** returns the eigenvalues and eigenvectors of a complex Hermitian (conjugate symmetric) or a real symmetric matrix; its parameters include:
  - Hermitian or real symmetric matrix whose eigenvalues and eigenvectors are to be computed: a
- **sklearn.metrics.roc_auc_score** computes ROC AUC:
  - True labels: y_true
  - Obtained scores: y_scores
- **scipy.io.loadmat** loads data from a MATLAB file and includes the following parameters:

  - file_name
  - mdict=None - dictionary in which to insert matfile variables
  - appendmat=True - to append the .mat extension to the end of the given filename

- **numpy.loadtxt** loads data from a text file and includes the following parameters:
  - file_name

- **networkx.draw** draws a graph G with Matplotlib; it includes the following parameters:
  - G - a networkx graph
  - node_color - an array with colors for each node of the graph (same order as when iterating through nodes of the graph)

- **networkx.union** combines 2 graphs in a single one; it includes the following parameters:
  - G, H - the 2 graphs
  - rename - node names of G and H can be changed by specifying the tuple rename=('G','H'); node "u" in G is then renamed "Gu" and "v" in H is renamed "Hv"

- **Graph.size** returns the number of edges or total of all edge weights; it can include the edge attribute that holds the numerical value used as a weight; ff None, then each edge has weight 1.

- **networkx.ego_graph** returns induced subgraph of neighbors centered at the specified node n within a given radius; it includes the following parameters:
  - G - a NetworkX Graph
  - n - a single node
  - radius - number, optional; includes all neighbors of distance $\leq$ radius from node n.

- **torch.Tensor.backward()** - computes the gradient of current tensor

- **torch.optim.Optimizer.step()** - performs a single optimization step to update parameters.

- **torch.optim.Optimizer.zero_grad()** - resets the gradients of all optimized tensors

- **torch.optim.Adam** - implements Adam algorithm and includes the following parameters:
  - params (iterable) – iterable of parameters to optimize
  - lr – learning rate (default: 1e-3)

2 Exercises

2.1 Ex. 1

1. In the first exercise you will implement a version of the OddBall algorithm discussed in the course. You will first load the dataset from the ca-AstroPh.txt file (each line contains an edge of the graph) and build the corresponding undirected graph using **networkx.Graph**. Each appearance of an edge in the dataset will increase the weight of the edge by one (you can store the weight for each edge with a key using **G[node1][node2]["weight"] = weight**).

2. Extract the four features from the egonet (subgraph consisting of all neighbours) corresponding to each node:

   - N_i - number of neighbors
   - E_i - number of edges in egonet i
   - W_i - total weight of egonet i
   - λ_w, i - principal eigenvalue of the weighted adjacency matrix of egonet i

   Store the features in the Graph object using **nx.set_node_attributes**

3. Compute the anomaly score for each node by fitting a **LinearRegression** model (from sklearn.linear_model) with the logarithmic scale of the 2 features (E_i and N_i - which should detect near-cliques and stars). The anomaly score will be:

$$score_i = \frac{max(y_i, C{x_i}^{\theta})}{min(y_i, C{x_i}^{\theta})} \log(|y_i - C{x_i}^{\theta}| + 1)$$

   given the power-law equation $y = Cx^{\theta}$

4. Sort the scores of the nodes in descending order and draw the graph using $nx.draw()$. Using the *node_color* parameter for the specified function, draw the nodes corresponding to the biggest 10 scores with a different color. Use just the first 1500 rows from the file when generating the graph.

5. Modify the anomaly score as the sum of the normalized score that was computed earlier and the score obtained by LOF (for the pair of E_i and N_i features) and draw again the graph using a different color for the nodes corresponding to the biggest 10 scores.

2.2 Ex. 2

1. In this exercise we will generate some types of graph anomalies using **networkx** package. You will first generate a regular graph with 100 nodes

where each one will have a degree 3 using **networkx.random_regular_graph**. You will merge it with another graph that will contain 10 cliques with 20 nodes each. This one will be generated using **networkx.connected_caveman_graph()**. Both will be merged using **networkx.union()** and some random edges will be added in order to build a connected graph. Draw the resulted graph. Use the model developed in the first exercise to detect the first 10 nodes that are most probably part of a clique in the final merged graph (using E_i and N_i) and draw them using a different color.

2. Here you will generate some **HeavyVicinity** anomalies. You will generate a regular graph with 100 nodes where each one will have a degree 3 and another one that has 100 nodes and each member node has degree 5 (using **networkx.random_regular_graph**). You will merge them using **networkx.union()** and then you will assign weight 1 for each edge using **G.add_edge(edge[0], edge[1], weight=1)** while iterating over all the edges. You will pick 2 random nodes and you will add 10 to the weights of all the edges from their egonets (using **G[node1][node2]["weight"] += 10**). Then you will use the same model to draw with a different color the 4 nodes that have the greatest score (using W_i and E_i - which should detect the heavy vicinities).

2.3 Ex. 3

1. In this exercise we will design a Graph Autoencoder (GAE) capable of ranking anomalous nodes from an Attributed Graph (based on reconstruction error). For this, we will use **GCNConv** layers from **torch_geometric.nn**. The autoencoder will contain an Encoder that will encode both structure information and node attributes, and two separate decoders. One will reconstruct the attributes of the nodes from the latent representations of the encoder and the other one will reconstruct the adjacency matrix (from the same latent representations).

2. Load the ACM dataset (from ACM.mat file) and extract the attributes of the nodes ("Attributes" key), the adjacency matrix ("Network" key) and the labels of the nodes ("Label" key). You can convert the adjacency matrix from the sparse matrix format in edge list using **from_scipy_sparse_matrix()** function from **torch_geometric.utils.convert**

3. Design a graph autoencoder that subclasses the **torch.nn.Model** class. This will contain the encoder and the two decoder sub-models (these will also subclass **torch.nn.Model**). The encoder will contain:

    - 1 **GCNConv** layer which encodes the input samples in samples of size 128, followed by a **relu** activation
    - 1 **GCNConv** layer which encodes the output of the previous layer to 64-sized samples, followed by a **relu** activation

The attribute decoder will consist of the same 2 layers (but with reversed input/output size).

The structure decoder will contain one **GCNConv** layer (that will keep the dimension of the latent representations), followed by a **relu** activation. Then, the structure decoder will return the $Z@Z^T$ product (where Z is the output of the **relu** activation) in order to match the size of the adjacency matrix.

You have to define the **forward** method for each of the 3 models (in order to define the computations performed when data is passed through the network).

4. You have to define the following custom loss function that takes as input the original attributes of the nodes - X, the reconstructed ones - $\hat{X}$, the original adjacency matrix - A, the reconstructed one - $\hat{A}$ and the α parameter which balances the importance of attribute and structure reconstruction (for alpha you will use a value of 0.8):

$$\mathcal{L} = \alpha\|X - \hat{X}\|_F^2 + (1-\alpha)\|A - \hat{A}\|_F^2$$

5. You have to define your training procedure. First you will define your optimizer (**Adam**, with a learning rate of 0.004) and then you will perform the necessary computations for each epoch:

   - reset the gradients using **optimizer.zero_grad()**
   - pass data through the autoencoder and get the reconstructions for attributes and adjacency matrix
   - compute the loss
   - compute the gradients using **backward()** method
   - update the network weights using **optimizer.step()**
   - at every 5 epochs compute the ROC AUC score for the data based on the scores represented by the reconstruction errors