# Laboratory 2.2: Source Localization

**Objectives:**

- Create a graph using a stochastic block model (SBM).
- Implement a graph neural network (GNN) to determine which community originated a certain graph signal.
- Analyse the effect of varying levels of noise and number of communities.

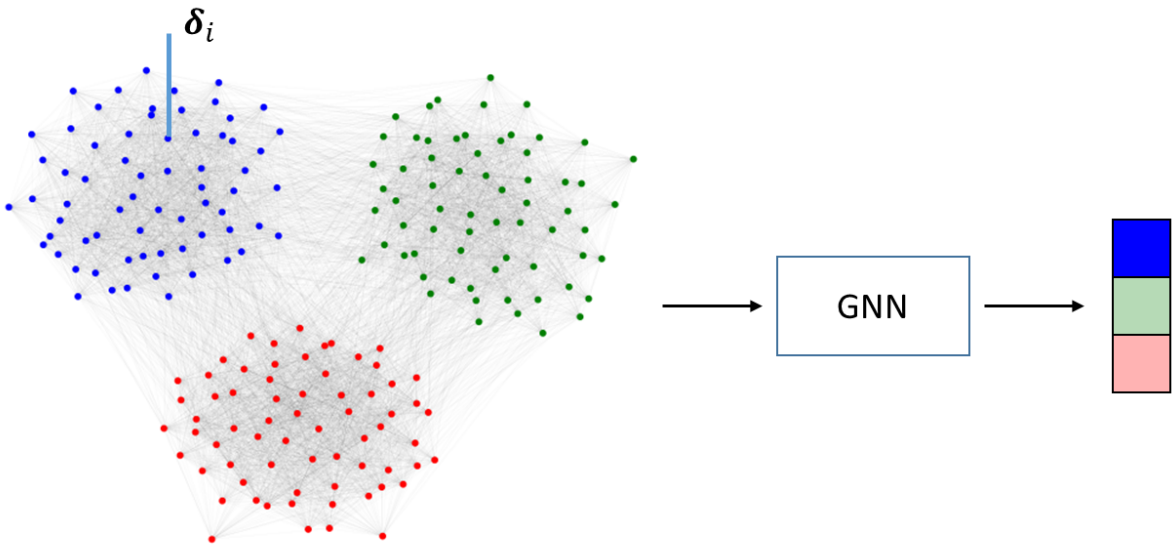
**Completion requirements:**

By the end of this notebook, you should have:

- Implemented all code cells for:
  - defining a GNN-model.
  - training the GNN-model
  - assesing the performance of the GNN-model.
- Answered the analysis questions on each section.

## Background

Graphs such as social networks can be divided in communities based on common traits. In several tasks, we are interested in finding the source of the information. In other words, which community is responsible for generating a fake news, outbreak, epidemics, or rumor. To determine where a signal originated from we can use graph based tools such as GNNs. In the source localization problem, we have to estimate which community $c$ originated the diffusion, by observing graph signals and knowing the topology.



# ⛔ Problem definition

In this notebook, you are asked to create a graph neural network (GNN) to determine the community which originated a certain graph signal. This problem is a supervised graph signal classification given a training set of input-output pairs $(\mathbf{x}, c)$, where $c$ is the community that has originated the diffusion and $\mathbf{x}$ is the graph signal. The output can be given by a one-hot vector on the number of communities, and we can use a cross-entropy loss for training.

Dataset details

In this laboratory, we consider a synthetic dataset created via a stochastic block model graph with $C$ communities. We generate the data by diffusing unitary signals from different source nodes and sampling it after a certain number of diffusion steps. Your task will then be to determine from which community the signal is originated from.

# 📒 Libraries

In [ ]:

```
import torch
import networkx as nx
import numpy as np
from torch.nn.functional import one_hot

from torch_geometric.utils import from_networkx
```

## Create a graph using a stochastic block model (SBM)

First of all you need to create a function that generates a graph with $C$ number of communities using a stochastic block model. The function should have as arguments the number of nodes of the graph, the number of communities, and the probabilties of creating an edge inside a community and among different communities

In [ ]:

```
def generate_sbm_graph(num_nodes, num_communities=2,
                       intra_prob=0.8, inter_prob=0.1, seed=42):
  """
  Description:

  This function generates a graph using the Stochastic Block Model (SBM),
  which assumes that nodes in a graph belong to one or more communities,
  and edges between nodes are generated probabilistically based on the community assignme

  Parameters:

  num_nodes (int): The total number of nodes in the generated graph.
  num_communities (int): The desired number of communities in the generated graph.
  intra_prob (float): The probability of an edge existing between two nodes within the sa
  inter_prob (float): The probability of an edge existing between two nodes in different
  seed (int): The seed for the random number generator used in generating the graph.

  Returns:

  adjacency_matrix (numpy.ndarray): A 2D array representing the adjacency matrix of the g
    The entry at index (i, j) represents the presence (1) or absence (0) of an edge betwe
  labels (numpy.ndarray): A 1D numpy array representing the community assignments of each
    The entry at index i represents the community assignment of node i, which is an integ

  Note:
  Be sure to balance the number of nodes in each community.
  """
  np.random.seed(seed)

  ...

  return adjacency_matrix, labels
```

In [ ]:

```
num_nodes = 200 # Number of nodes
num_communities = 3 # Number of communities

intra_prob = 0.7 # Probability of drawing edges intra communities
inter_prob = 0.2 # Probability of drawing edges inter communities

adjacency_matrix, labels = generate_sbm_graph(num_nodes, num_communities, intra_prob, int
```

In [ ]:

```
color_mapping=['blue','red','green','purple','pink','black']
color_map = [color_mapping[i] for i in labels]

# Convert adjacency matrix to networkx graph
graph = nx.from_numpy_array(adjacency_matrix)

nx.draw(graph,
        node_color=color_map,
        node_size=10, width=0.01)
```

### Questions

- Can you clearly identify the communities? If not, why?

**Answers:**

When the intra and inter probabilities are close to each other the communities start to merge. Otherwise, the networkx plotting tool should clearly highlight the different communities.

## Diffuse the signal from a source node and sample it at a random time

After generating the SBM graph, you are asked to diffuse a unitary node signal following this equation:

$$\mathbf{x}_t = \mathbf{S}^t \boldsymbol{\delta}_i + \boldsymbol{\epsilon}_t$$

where $\mathbf{x}_t$ is the diffused graph signal $\mathbf{S}$ is the graph shift operator, $\boldsymbol{\delta}_i$ is a graph signal that has a 1 in node $i$ and 0 elsewhere, and $\boldsymbol{\epsilon}_t$ is a Gaussian noise with zero mean and std. deviation $\sigma$. This equation propagates the graph signal $\boldsymbol{\delta}_i$ for $t$ diffusion steps.

Write a function that implements the above equation, given a graph, a source node, the noise standard deviation, and the number of diffusion steps.

In [ ]:

```
def generate_diffusion(adjacency_matrix, source_node, n_diffusions, noise_std=0.01, seed=
  """
  Description:
  This function generates a dataset of diffused signals given a source node on a graph re
  The diffusion process is modeled as iteratively multiplying the signal vector with the
  and adding Gaussian noise at each iteration.

  Parameters:

  adjacency_matrix (numpy.ndarray): A 2D numpy array representing the adjacency matrix of
  source_node (int): Index of the source node from which the diffusion process starts.
  n_diffusions (int): Number of diffusion steps to perform.
  noise_std (float): Standard deviation of the Gaussian noise added at each diffusion ste
  seed (int): The seed for the random number generator used in generating the Gaussian no

  Returns:

  signal (torch.Tensor): A 1D tensor representing the final diffused signal after the spe
    The length of the array is equal to the number of nodes in the graph,
    and the entry at index i represents the diffused signal value of node i.

  Note:

  The adjacency matrix is normalized by dividing it by the maximum eigenvalue to avoid ex
  """
  np.random.seed(seed)

  # Normalize the adjacency matrix to avoid exploding values
  e, V = np.linalg.eigh(adjacency_matrix)
  e_max = np.max(e)
  adjacency_matrix = adjacency_matrix / e_max

  ...

  return signal # [N]
```

## Create database

Create a function that defines different instances of diffused graph signals, using for each instance a random source node and a random number of diffusion steps.

Afterwards, generate the dataset and split it into training, validation, and testing sets.

In [ ]:

```
def create_dataset(total_samples, adjacency_matrix, maximum_diffusion_steps=10, noise_std
  """
  Description:
  This function generates a dataset of diffused signals for a specified number of samples
  The diffusion process starts from randomly selected source nodes and is performed for a
  The resulting dataset is returned as tensors of input signals (x) and corresponding lab

  Parameters:

  total_samples (int): The total number of samples to generate in the dataset.
  adjacency_matrix (numpy.ndarray): A 2D array representing the adjacency matrix of the g
  maximum_diffusion_steps (int): The maximum number of diffusion steps to perform for eac
    The actual number of diffusion steps is randomly selected for each sample from the ra
  noise_std (float): The standard deviation of the Gaussian noise added at each diffusion

  Returns:

  all_x (torch.Tensor): A 3D tensor representing the input signals (diffused signals) in
  all_y (torch.Tensor): A 2D tensor representing the labels (one-hot encoded) correspondi
  """
  all_x = []
  all_y = []

  ...

  return all_x, all_y
```

In [ ]:

```
train_size  = 200 # Number of training samples
val_size    = 50 # Number of validation samples
test_size   = 50 # Number of testing samples

total_samples = train_size + val_size + test_size

all_x, all_y = create_dataset(total_samples, adjacency_matrix, maximum_diffusion_steps=10
```

In [ ]:

```
def create_train_val_test(graph, all_x, all_y, train_size, val_size, test_size):
  """
  This function splits the generated dataset into training, validation, and testing datas
  based on the specified sizes.
  It creates separate datasets for each split, where each dataset consists of data object
  input signals (x) and labels (y) based on the generated diffused signals.

  Note:

  The training, validation, and testing datasets are returned as lists of PyTorch Geometr
  """
  train_dataset = []
  val_dataset = []
  test_dataset = []

  ...

  return train_dataset, val_dataset, test_dataset
```

In [ ]:

```
train_dataset, val_dataset, test_dataset = create_train_val_test(graph, all_x, all_y, tra
```

**Dataset sample**

This is how one sample of your dataset should look like:

- **edge_index:** Graph connectivity in COO format with shape [2, num_edges]
- **x:** Input node features with shape [Nodes, Node features]. The number of node features is 1 for for our case and it represents the value of the diffused graph signal
- **y:** Output graph signal, i.e. target feature. Shape [1, num_communities].

In [ ]:

```
train_dataset[0]
```

# 💻 Model definition and training

**Instructions:** Define a GNN model class, instantiate it, and train it.

The GNN takes as input the diffused node signal and returns the estimate of which community originated it.

In [ ]:

```
from torch_geometric.nn import ChebConv
from torch_geometric.loader import DataLoader

import torch.nn as nn
```

In [ ]:

```
class Net(torch.nn.Module):
    """
    Graph neural network (GNN) model for community detection.

    Args:
        num_communities (int): Number of communities in the graph.
        num_nodes (int): Number of nodes in the graph.
        hid_features (int): Number of hidden features in the GNN layers.
    """
    def __init__(self, num_communities, num_nodes, hid_features=32):
        super(Net, self).__init__()

        ...

    def forward(self, data):

        ...

      return x
```

**Model instantiation**

In [ ]:

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

model = Net(num_communities, num_nodes, hid_features=16).to(device)
```

## Train a Graph Neural Network

Train your GNN model.

Define the training and evaluation functions.

Pay attention to which loss function you are going to use.

**Define the training and evaluation functions.**

In [ ]:

```
def train_epoch(model, loader, optimizer, device='cpu'):
  """
    Trains a neural network model for one epoch using the specified data loader and optim

    Args:
        model (nn.Module): The neural network model to be trained.
        loader (DataLoader): The PyTorch Geometric DataLoader containing the training dat
        optimizer (torch.optim.Optimizer): The PyTorch optimizer used for training the mo
        device (str): The device used for training the model (default: 'cpu').

    Returns:
        float: The mean loss value over all the batches in the DataLoader.

    """
    model.to(device)
    model.train() # specifies that the model is in training mode

    ...

    return loss
```

In [ ]:

```
@torch.no_grad()
def evaluate_epoch(model, loader, device='cpu'):
    """
    Evaluates the performance of a trained neural network model on a dataset using the sp

    Args:
        model (nn.Module): The trained neural network model to be evaluated.
        loader (DataLoader): The PyTorch Geometric DataLoader containing the evaluation d
        device (str): The device used for evaluating the model (default: 'cpu').

    Returns:
        float: The mean loss value over all the batches in the DataLoader.

    """
    model.to(device)
    model.eval() # specifies that the model is in evaluation mode

    ...

    return loss, accuracy
```

### Define the DataLoaders and the optimizer

In [ ]:

```
train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=16, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=16, shuffle=False)

optimizer = torch.optim.Adam(model.parameters(), lr=0.1, weight_decay=5e-4)
```

### Train the model

In [ ]:

```
...
```

# 📈 Results

**Visualize loss and testing results**

Plot the training and validation losses as a function of the epochs and analyse the performance of the network.

## 🕵️ Questions

- What is the final testing accuracy? How does it relate to the training performance of the network?

**Answers**

The final testing accuracy is around 100%, which is equal to the validation loss, implying that the model was correctly trained.

# Effects of noise level

Generate data with increased noise levels and re-train your model.

Plot test accuracy versus noise for at least 5 noise levels.

## 🕵️ Questions

- What do you conclude if the noise level increases?

**Answers:**

As the noise level increases, the test accuracy decreases.

In [ ]:

```
plt.scatter(test_accs.keys(), test_accs.values())
plt.xlabel("Noise level")
plt.ylabel("Test accuracy");
# plt.xscale('log')
```

# Effect on number of commuities

Generate data with increased number of communities and re-train your model.

Plot test accuracy versus number of communities for at least 5 values.

## Questions

- How is the performance affected by the number of communities?

**Answers**

The test accuracy should decrease with the number of communities.

In [ ]:

```
plt.scatter(test_accs.keys(), test_accs.values())
plt.xlabel("Number of communities")
plt.ylabel("Test accuracy")
```