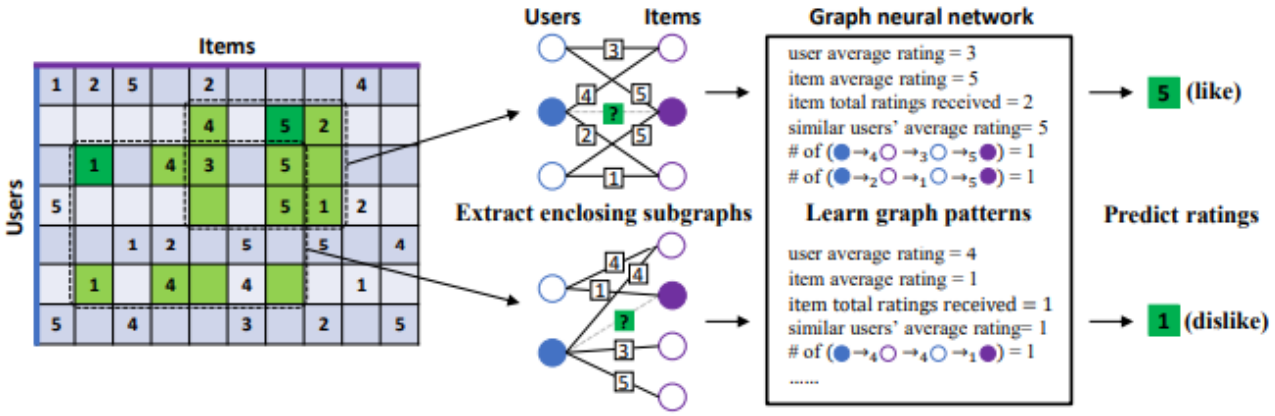


Unlocking Graph-Based Machine Learning: Exploring Inductive Graph-Based Matrix Completion (IGMC)

Graph-based machine learning (GraphML) is a frontier in AI, harnessing the power of graph structures to model relationships between entities. In this blog post, we'll dive into a practical implementation of **Inductive Graph-based Matrix Completion (IGMC)**, unpacking the methodology and code from a notebook.

If you're familiar with machine learning but new to graph-based approaches, this guide will bridge the gap, showing how IGMC can predict missing entries in matrices by modeling interactions as graphs.

---

1. What is Matrix Completion?

$$M = \begin{bmatrix} 5 & ? & 3 \\ 4 & 2 & ? \\ ? & 1 & 5 \end{bmatrix}$$

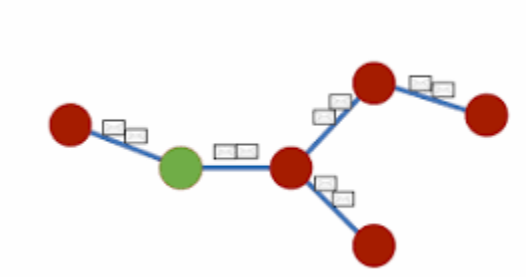
Matrix completion is a class of problems that aims to predict missing values in a matrix, such as user-item interactions in recommender systems. For example:

- Predicting user ratings for movies (Netflix problem).
- Estimating missing links in social networks.

Instead of viewing this as a matrix operation, IGMC reframes it as a **graph problem**.

---

2. Why Graphs for Matrix Completion?



Graph-based models represent data as nodes (entities) and edges (relationships). IGMC leverages this representation to:

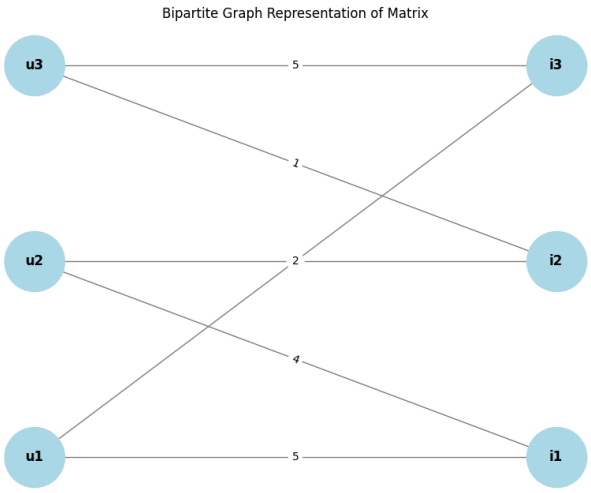
- **Model complex relationships:** Beyond rows and columns, graphs capture intricate connections.
- **Generalize effectively:** Instead of fitting only the given data, it can inductively predict unseen patterns.

---

3. Overview of IGMC

IGMC is a method for predicting missing matrix entries using graph neural networks (GNNs). Here's how it works:

- **Transform data into a graph:** Represent rows, columns, and known values as a bipartite graph.



- **Subgraph extraction:** For each target prediction, extract a local subgraph of relevant interactions.
- **Graph learning:** Use GNNs to learn embeddings and predict the missing values.

This inductive approach means the model can generalize to unseen nodes during inference.

---

4. Key Components of the Notebook

The notebook implements IGMC step by step. Here's an outline of the pipeline:

1. **Installation of the libraries**

```
import torch

!pip uninstall torch-scatter torch-sparse torch-geometric torch-cluster  --y
!pip install torch-scatter -f https://data.pyg.org/whl/torch-{torch.__version__}.html
!pip install torch-sparse -f https://data.pyg.org/whl/torch-{torch.__version__}.html
!pip install torch-cluster -f https://data.pyg.org/whl/torch-{torch.__version__}.html
!pip install git+https://github.com/pyg-team/pytorch_geometric.git
```

The libraries mentioned in graph machine learning (Graph ML) are part of the PyTorch Geometric (PyG) ecosystem. Each plays a distinct role in implementing and optimizing graph neural networks (GNNs). Here's an explanation:

1. torch-scatter

- **Purpose**: Implements efficient scatter and segment operations, which are essential for operations on graphs.
  Use Case: It is used to perform reductions (e.g., summation, mean, max) on tensors along specified indices, which is a fundamental operation when aggregating information from graph neighborhoods in GNNs.

2. torch-sparse

- **Purpose**: Provides sparse matrix operations tailored for GNNs.
  Use Case: It enables efficient computation with sparse tensors, such as adjacency matrices. This is crucial because graph data is inherently sparse, and using sparse representations improves memory efficiency and computation speed.

3. torch-cluster

- **Purpose**: Implements clustering algorithms for point clouds and graphs.
  Use Case: It is often used for neighborhood sampling, constructing k-nearest neighbors (k-NN) graphs, and other clustering-related tasks necessary for GNN preprocessing or during training.

4. torch-geometric

- **Purpose**: The main library for building and training GNNs.
  Use Case: Provides a high-level API for defining and training GNNs, with pre-built layers, models, and utilities for working with graph data (e.g., data loading, batching, transformations).

2. Importing libraries

```
import torch
from torch.nn import Linear
import torch.nn.functional as F
from torch.optim import Adam
from torch_geometric.data import DataLoader
from torch_geometric.nn import RGCNConv
from torch_geometric.utils import dropout_adj
from util_functions import *
from data_utils import *
from preprocessing import *
```

1. DataLoader (from torch_geometric.data)

- **Purpose**: Handles batching, shuffling, and iterating over graph datasets.
  Use Case: In graph machine learning, datasets often consist of multiple graphs or subgraphs. The DataLoader ensures these graphs are efficiently loaded and processed in batches during training or evaluation.
- **Example**: If you have a dataset of multiple molecular graphs, the DataLoader can provide mini-batches of these graphs to the model.

2. RGCNConv (from torch_geometric.nn)

- **Purpose**: Implements the Relational Graph Convolutional Network (RGCN) layer.
  Use Case: RGCN is designed for handling graphs with multiple types of edges (relations). It is widely used in tasks like knowledge graph completion or social network analysis where edges have diverse semantics.
- **Key Features:**
  Supports heterogeneous graphs with multiple edge types.
  Aggregates neighbourhood information while respecting edge types.

3. dropout_adj (from torch_geometric.utils)

- **Purpose**: Applies dropout to the adjacency matrix of a graph, effectively removing some edges randomly during training.
- **Use Case**: Edge dropout is a regularization technique to prevent overfitting in GNNs by encouraging the model to generalize better to unseen edges.

This can be particularly useful in sparse graphs where certain edges might dominate the learning process.

- **Example**: During training, some edges are dropped randomly to simulate variations in graph structure, similar to how traditional dropout works for nodes or layers in neural networks.

These imports provide essential tools for implementing and training relational graph convolutional networks (RGCNs) while also managing graph data efficiently and applying regularization to improve model performance.

3. Data Loading and Preprocessing

We use the MovieLens 100K dataset, a popular benchmark for recommendation systems, to model user-movie interactions. Let's break down the data preparation process in detail.

1. Loading the Dataset

The dataset is split into training, validation, and testing sets using the provided function:

```
(u_features, v_features, adj_train, train_labels, train_u_indices, train_v_indices, val_labels,
val_u_indices, val_v_indices, test_labels, test_u_indices, test_v_indices, class_values
) = load_official_trainvaltest_split('ml_100k', testing=True)
```

- **`u_features` and `v_features`**: User and movie features. For this setup, they're set to `None`, as we're not using additional node features in this experiment.
- **`adj_train`**: The adjacency matrix representing the interaction graph for training. It encodes the relationships between users and movies.
- **`train_labels`, `val_labels`, and `test_labels`**: The interaction labels (e.g., ratings) for the respective splits.
- **`train_u_indices` and `train_v_indices`**: User and movie indices for the training dataset, indicating which users interacted with which movies. Similar indices are provided for validation (`val_u_indices`, `val_v_indices`) and testing (`test_u_indices`, `test_v_indices`).
- **`class_values`**: The unique classes (ratings) present in the dataset.

---

2. Constructing the Datasets

Two dataset classes are instantiated, one for training and one for testing:

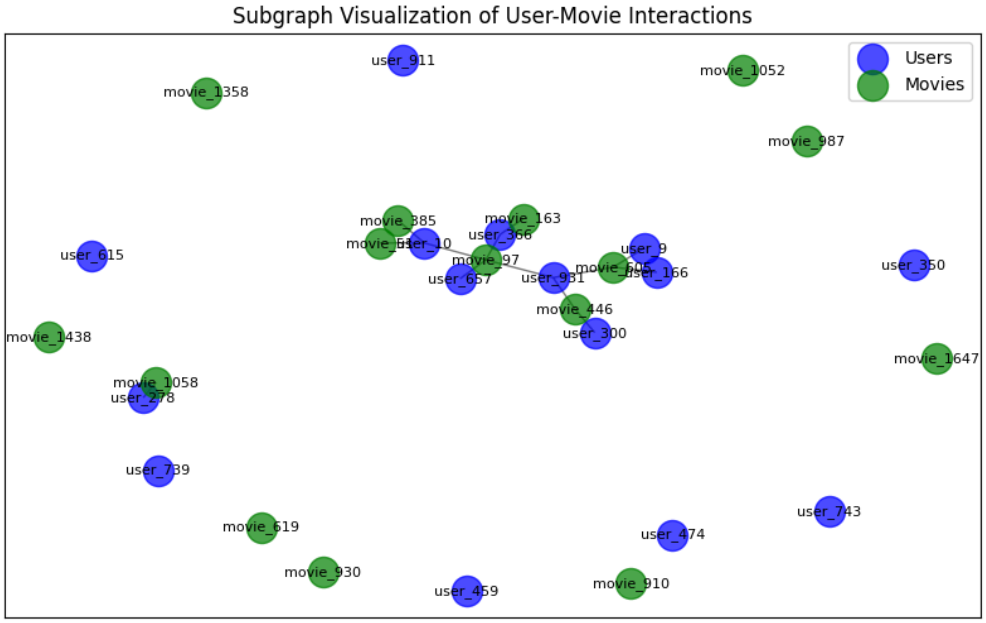
```
train_dataset = eval('MyDynamicDataset')(root='data/ml_100k/testmode/train', A=adj_train,
    links=(train_u_indices, train_v_indices), labels=train_labels, h=1, sample_ratio=1.0,
    max_nodes_per_hop=200, u_features=None, v_features=None, class_values=class_values)
test_dataset = eval('MyDataset')(root='data/ml_100k/testmode/test', A=adj_train,
    links=(test_u_indices, test_v_indices), labels=test_labels, h=1, sample_ratio=1.0,
    max_nodes_per_hop=200, u_features=None, v_features=None, class_values=class_values)

len(train_dataset), len(test_dataset)
```

- **`MyDynamicDataset` and `MyDataset`**: These are custom dataset classes tailored to generate graph samples dynamically or use pre-sampled graphs for training and testing.
- **Key Parameters**:
  - **`A`**: The adjacency matrix for the user-movie interaction graph.
  - **`links`**: A tuple of user and movie indices representing interactions (e.g., user 0 rated movie 42).
  - **`labels`**: Interaction labels (ratings).
  - **`h`**: The hop count for subgraph sampling. Here, a value of `1` means only immediate neighbors are considered.
  - **`sample_ratio`**: The ratio of neighbors sampled for each node. A value of `1.0` means all neighbors are included.
  - **`max_nodes_per_hop`**: Limits the maximum number of nodes sampled per hop to ensure efficiency.
  - **`u_features` and `v_features`**: Since these are `None`, we are not incorporating additional node-level features into the model.

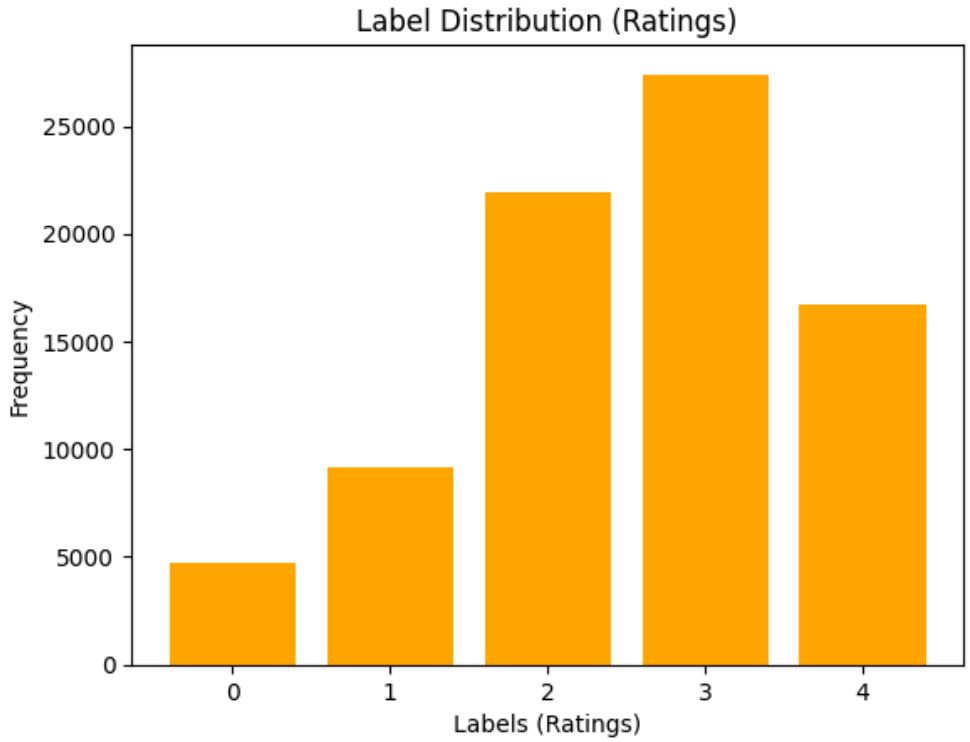
---

3. Data visualisation

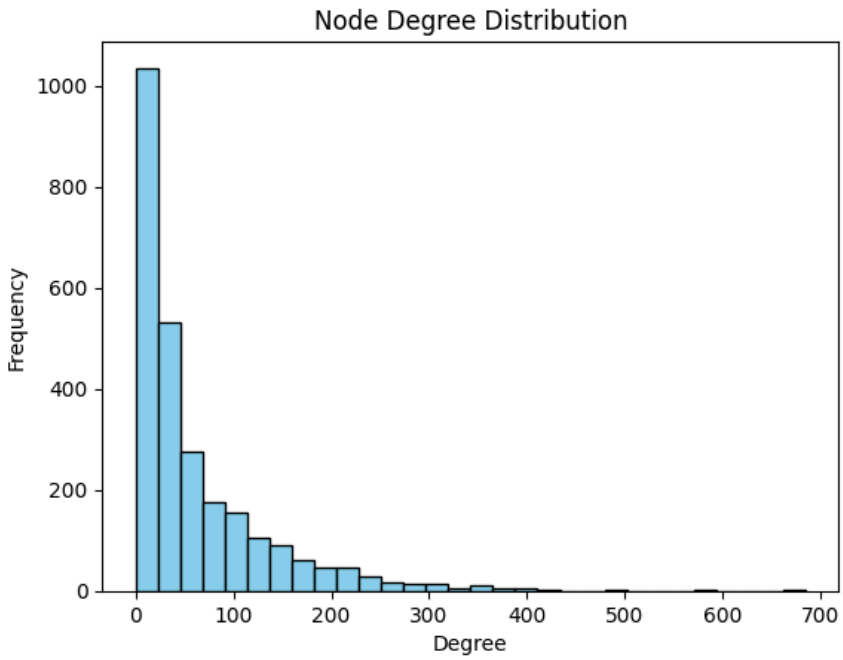
1. A subgraph visualisation:



2. Label distribution:



3. Node degree distribution:



3. Creating DataLoaders

Finally, the datasets are wrapped into PyTorch `DataLoader` objects for efficient batch processing during training and testing:

```
train_loader = DataLoader(train_dataset, BATCH_SIZE, shuffle=True, num_workers=2)
test_loader = DataLoader(test_dataset, BATCH_SIZE, shuffle=False, num_workers=2)
```

- **`train_loader`**: Loads batches from the `train_dataset`, shuffling the data to ensure randomness during training.
- **`test_loader`**: Loads batches from the `test_dataset` in sequential order for evaluation.
- **Key Parameters**:
  - **`BATCH_SIZE`**: The number of samples processed in a single batch.
  - **`shuffle`**: Ensures randomness during training but is disabled for testing.
  - **`num_workers`**: Specifies the number of CPU workers for parallel data loading.

---

4. Insights into the Data

- **Dynamic Graph Sampling:** The custom dataset classes allow for on-the-fly sampling of subgraphs centered around user-movie interactions. This is particularly useful for large graphs where storing all subgraphs might be infeasible.
- **Graph Structure:** Each interaction is represented as a graph with the user and movie as central nodes. Neighbors are sampled up to one hop away, providing a local context for each interaction.
- **Scalability:** Parameters like `sample_ratio` and `max_nodes_per_hop` are essential for managing computational complexity, especially for dense graphs with many connections.

4. Graph Neural Network (GNN) Implementation

```
class IGMC(torch.nn.Module):
    def __init__(self):
        super(IGMC, self).__init__()
        self.rel_graph_convs = torch.nn.ModuleList()
        self.rel_graph_convs.append(RGCNConv(in_channels=4, out_channels=32, num_relations=5, num_bases=4))
        self.rel_graph_convs.append(RGCNConv(in_channels=32, out_channels=32, num_relations=5, num_bases=4))
        self.rel_graph_convs.append(RGCNConv(in_channels=32, out_channels=32, num_relations=5, num_bases=4))
        self.rel_graph_convs.append(RGCNConv(in_channels=32, out_channels=32, num_relations=5, num_bases=4))
        self.linear_layer1 = Linear(256, 128)
        self.linear_layer2 = Linear(128, 1)

    def reset_parameters(self):
        self.linear_layer1.reset_parameters()
        self.linear_layer2.reset_parameters()
        for i in self.rel_graph_convs:
            i.reset_parameters()

    def forward(self, data):
        num_nodes = len(data.x)
        edge_index_dr, edge_type_dr = dropout_adj(data.edge_index, data.edge_type, p=0.2, num_nodes=num_nodes, training=self

        out = data.x
        h = []
        for conv in self.rel_graph_convs:
            out = conv(out, edge_index_dr, edge_type_dr)
            out = torch.tanh(out)
            h.append(out)
        h = torch.cat(h, 1)
        h = [h[data.x[:, 0] == True], h[data.x[:, 1] == True]]
        g = torch.cat(h, 1)
        out = self.linear_layer1(g)
        out = F.relu(out)
        out = F.dropout(out, p=0.5, training=self.training)
        out = self.linear_layer2(out)
        out = out[:,0]
        return out

model = IGMC()
```

Activate Wir
Go to Settings to

The IGMC class is an implementation of a graph neural network (GNN) model based on Relational Graph Convolutional Networks (RGCNs) for graph-related tasks. Let's break down the implementation step by step:

1. Model Architecture

The model architecture is defined in the __init__ method. Here's a breakdown:

a. self.rel_graph_convs

- **Definition**: A list of 4 RGCNConv layers.
- **Purpose**: Each layer performs a relational graph convolution. These layers capture neighborhood information while considering the relational semantics (i.e., edge types).
- **Parameters**:
    - in_channels: The number of input features for each node.
    - out_channels: The number of output features for each node.
    - num_relations: The number of unique edge types in the graph.
    - num_bases: A hyperparameter to reduce the number of parameters in edge-type-specific weight matrices.
    - Flow: The feature dimension progresses as follows:
      Layer 1: Input features (4) → Output features (32)
      Layers 2–4: Output features remain (32).

2. Parameter Initialization

The reset_parameters method initializes the weights of all layers:

- Resets the linear layers using their internal reset methods.
- Resets each RGCNConv layer to its initial state.

3. Forward Pass

The forward method defines the computation for a batch of graph data (data).

a. Graph Dropout

- **dropout_adj**: Randomly drops edges and their associated types from the adjacency matrix (edge_index) and edge type matrix (edge_type).
- **Purpose**: This regularization step prevents overfitting and simulates variations in graph structures during training.

b. Node Feature Propagation

- **Initialization**: out = data.x (node features from the input data).
- **Propagation Through RGCNs:** Iteratively applies each RGCNConv layer.
- **Activation**: After each layer, applies the hyperbolic tangent activation function (torch.tanh) to introduce non-linearity.
- **Feature Storage**: Stores the outputs of each RGCN layer in h.

c. Feature Aggregation

- **Concatenation**: Combines the outputs of all RGCN layers into a single tensor h with shape [num_nodes, total_features].
- **Masking**: Splits h into two subsets of features based on a condition:
  h[data.x[:, 0] == True]: Nodes satisfying condition 1.
  h[data.x[:, 1] == True]: Nodes satisfying condition 2.
- **Concatenation**: Combines these subsets into g.

d. Linear Layers

Passes the aggregated features g through two fully connected layers with:

- **Activation**: Applies ReLU after the first layer.
- **Dropout**: Regularizes with dropout (p=0.5) before the second linear layer.

4. Loss Computation :

We used:

- **MSE**: for training
- **MSE &RMSE:** for evaluation