* We import several libraries.
* MinMaxScaler from sklearn.preprocessing is responsible for scaling our data between 0 and 1 to normalize it. torch.nn contains layers and functions to define the neural network structure. torch\_geometric is used for working with graph-based neural networks, with GCNConv providing graph convolution layers and GATConv offering graph attention layers. scatter\_mean is used to aggregate node features.
* load the training and test datasets from CSV files into pandas DataFrames using pd.read\_csv(). We specify low\_memory=False to ensure efficient reading of large datasets. For the test dataset, skip the first row using skiprows=[0] it may contain irrelevant metadata.
* The preprocess\_data() function cleans the dataset by first dropping unnecessary columns . We check for a 'Time' column, and if it exists, then dropped, since we don't need it.
* Within the custom\_convert() function, we handle numeric strings containing 'K' (for thousand) or 'M' (for million) by replacing these suffixes with the respective number of zeros. This function converts the string-based values into numerical format.
* The 'Number of Users' column is checked next. If it exists, it is converted into numeric form. We also filter out any rows where the number of users is zero. Similarly'Average Response Time' column and remove any rows where the response time is zero.
* Then, using apply(), we apply custom\_convert() to all columns of type object to ensure numeric values are properly formatted.
* Once the data is preprocessed, we normalize both the training and test data using the MinMaxScaler. This step is crucial because features might have different ranges, and normalization brings all values into a consistent range between 0 and 1, helping the model train more effectively. After normalizing the data, we also normalize the target column ('Average Response Time') using another MinMaxScaler.
* Next, we convert the normalized data into PyTorch tensors.
* We create a function sliding\_window\_view() to implement a sliding window mechanism for time-series data. This function generates windows of consecutive time steps, which the model will use as input. The time window is set to 10, meaning that for each prediction, the model will look at the previous 10 time steps.
* The target variable, 'Average Response Time', is then extracted from the normalized training data as a tensor and reshaped into a column vector.
* We then define the graph structure. We calculate the number of nodes, which is equal to the number of time steps in each window. The edge\_index is constructed to create a fully connected graph, where each node (time step) is connected to every other node, except itself. This graph will serve as the structure for our graph neural network.
* We also calculate the batch size, which corresponds to the number of sliding windows, and create a batch\_index that maps each node to its respective graph. This is necessary for batching data during training.
* Next, we create Data objects using PyTorch Geometric. These objects represent the graph-based data, containing node features, edge indices, target values, and batch indices for both training and test data. These Data objects will be fed into the graph neural network for training and evaluation.
* We define the GNN model using a custom class GNNModel. This model starts with a graph convolutional layer (GCNConv) to learn node representations from neighbors, followed by a graph attention layer (GATConv) that applies attention mechanisms to focus on the most important nodes in the graph. Finally, we use fully connected (linear) layers to transform the learned representations and predict the output (in this case, the future average response times).
* Once the model is defined, we proceed with training it using gradient descent. The model is trained for multiple epochs to minimize the loss between the predicted values and the actual values from the training data. The training loop computes the loss, backpropagates the error, and updates the model weights using the optimizer.
* Finally, once the model is trained, we use it to make predictions on the test data. These predictions are then scaled back to the original range (using inverse\_transform on the target scaler), so they match the units of the original dataset (e.g., seconds for response time).