GNNExplaner(torch.nn.Module):

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

Args:
        model (torch.nn.Module): The GNN module to explain.
        epochs (int, optional): The number of epochs to train.
            (default: :obj:`100`)
        lr (float, optional): The learning rate to apply.
            (default: :obj:`0.01`)
        num_hops (int, optional): The number of hops the :obj:`model` is
            aggregating information from.
            If set to :obj:`None`, will automatically try to detect this
            information based on the number of
            :class:`~torch_geometric.nn.conv.message_passing.MessagePassing`
            layers inside :obj:`model`. (default: :obj:`None`)
        return_type (str, optional): Denotes the type of output from
            :obj:`model`. Valid inputs are :obj:`"log_prob"` (the model
            returns the logarithm of probabilities), :obj:`"prob"` (the
            model returns probabilities), :obj:`"raw"` (the model returns raw
            scores) and :obj:`"regression"` (the model returns scalars).
            (default: :obj:`"log_prob"`)
        feat_mask_type (str, optional): Denotes the type of feature mask
            that will be learned. Valid inputs are :obj:`"feature"` (a single
            feature-level mask for all nodes), :obj:`"individual_feature"`
            (individual feature-level masks for each node), and :obj:`"scalar"`
            (scalar mask for each each node). (default: :obj:`"feature"`)
        allow_edge_mask (boolean, optional): If set to :obj:`False`, the edge
            mask will not be optimized. (default: :obj:`True`)
        log (bool, optional): If set to :obj:`False`, will not log any learning
            progress. (default: :obj:`True`)
        **kwargs (optional): Additional hyper-parameters to override default
            settings in :attr:`~torch_geometric.nn.models.GNNExplainer.coeffs`.
```

```
    def explain_node(self, node_idx, x, edge_index, **kwargs):
        r"""Learns and returns a node feature mask and an edge mask that play a
        crucial role to explain the prediction made by the GNN for node
        :attr:`node_idx`.

        Args:
            node_idx (int): The node to explain.
            x (Tensor): The node feature matrix.
            edge_index (LongTensor): The edge indices.
            **kwargs (optional): Additional arguments passed to the GNN module.

        :rtype: (:class:`Tensor`, :class:`Tensor`)
        """
```

```
def visualize_subgraph(self, node_idx, edge_index, edge_mask, y=None,
                           threshold=None, edge_y=None, node_alpha=None,
                           seed=10, **kwargs):
        r"""Visualizes the subgraph given an edge mask
        :attr:`edge_mask`.

        Args:
```

```
            node_idx (int): The node id to explain.
                Set to :obj:`-1` to explain graph.
            edge_index (LongTensor): The edge indices.
            edge_mask (Tensor): The edge mask.
            y (Tensor, optional): The ground-truth node-prediction labels used
                as node colorings. All nodes will have the same color
                if :attr:`node_idx` is :obj:`-1`.(default: :obj:`None`).
            threshold (float, optional): Sets a threshold for visualizing
                important edges. If set to :obj:`None`, will visualize all
                edges with transparancy indicating the importance of edges.
                (default: :obj:`None`)
            edge_y (Tensor, optional): The edge labels used as edge colorings.
            node_alpha (Tensor, optional): Tensor of floats (0 - 1) indicating
                transparency of each node.
            seed (int, optional): Random seed of the :obj:`networkx` node
                placement algorithm. (default: :obj:`10`)
            **kwargs (optional): Additional arguments passed to
                :func:`nx.draw`.

        :rtype: :class:`matplotlib.axes.Axes`, :class:`networkx.DiGraph`
        """
```

Pytorch Graph Convolution functions, used by pkipf:

Load data – for this example they used cora dataset

adj, features, labels, idx_train, idx_val, idx_test = load_data()

where:

features = sp.csr_matrix(idx_features_labels[:, 1:-1], dtype=np.float32)

labels = encode_onehot(idx_features_labels[:, -1])

where:

idx_features_labels = np.genfromtxt("{}{}.content".format(path, dataset),

dtype=np.dtype(str))

also:

idx_train = range(140)

idx_val = range(200, 500)

idx_test = range(500, 1500), and

idx_train = torch.LongTensor(idx_train)

idx_val = torch.LongTensor(idx_val)

idx_test = torch.LongTensor(idx_test)

```
# build graph
    idx = np.array(idx_features_labels[:, 0], dtype=np.int32)
    idx_map = {j: i for i, j in enumerate(idx)}
    edges_unordered = np.genfromtxt("{}{}.cites".format(path, dataset),
                                    dtype=np.int32)
    edges = np.array(list(map(idx_map.get, edges_unordered.flatten())),
                     dtype=np.int32).reshape(edges_unordered.shape)
    adj = sp.coo_matrix((np.ones(edges.shape[0]), (edges[:, 0], edges[:, 1])),
                        shape=(labels.shape[0], labels.shape[0]),
                        dtype=np.float32)

    # build symmetric adjacency matrix
    adj = adj + adj.T.multiply(adj.T > adj) - adj.multiply(adj.T > adj)
```

General Pytorch ML/DL functions:

```
    loss_train = F.nll_loss(output[idx_train], labels[idx_train])
    acc_train = accuracy(output[idx_train], labels[idx_train])
```

where:

```
    output = model(features, adj)
```

Details:

F.nll_loss -

`torch.nn.functional.nll_loss(input, target, weight=None, size_average=None, ignore_index=-100, reduce=None, reduction='mean')`

- **input** – (N, C)(N,C) where *C = number of classes* or (N, C, H, W)(N,C,H,W) in case of 2D Loss, or (N, C, d_1, d_2, ..., d_K)$(N,C,d_1,d_2,...,d_K)$ where K \geq 1$K \geq 1$ in the case of K-dimensional loss. *input* is expected to be log-probabilities.
- **target** – (N)(N) where each value is 0 \leq \text{targets}[i] \leq C-1$0 \leq \text{targets}[i] \leq C-1$, or (N, d_1, d_2, ..., d_K)$(N,d_1,d_2,...,d_K)$ where K \geq 1$K \geq 1$ for K-dimensional loss.

- **weight** (*Tensor*, *optional*) – a manual rescaling weight given to each class. If given, has to be a Tensor of size *C*

```
def test(model, data):
  model.eval()
  logits, accs = model(data.x, data.edge_index, data), []
  for _, mask in data('train_mask', 'test_mask'):
    pred = logits[mask].max(1)[1]
    acc = pred.eq(data.y[mask]).sum().item() / mask.sum().item()
    accs.append(acc)
  return accs
```

pred.eq

When looking at the example for gnnexplainer_cora, found in repos folder:

Equivalencies between two repos:

Net() returns same thing as GCN() in kipf implementation

In the example, logits, accs = model(data.x, data.edge_index, data, [])   data can be None

Kipf: output = model(features, adj)

In the example, loss = F.nll_loss(log_logits[data.train_mask], data.y[data.train_mask])

Kipf: loss_val = F.nll_loss(output[idx_val], labels[idx_val])

Features = x = data.x

Adj = data.edge_index =

Data.y = labels

Log_logits[data.train_mask] = output[idx_val]

Data.y[data.train_mask] = labels[idx_val]

Also Need to know:

Edge_mask

Where:

Example:

node_feat_mask, edge_mask = explainer.explain_node(node_idx, x, edge_index)

kipf:

ax, G = explainer.visualize_subgraph(node_idx, adj, edge_mask, y=data.y)

hence: ax = node_feat_mask

G = edge_mask

Edge_weight

In example:

Edge_weight is not used in explainer.explain_node

In pytorch_geometric example:

explaniner.explain_node(node_idx, features, adj,

edge_weight=edge_weight)

where:

```
log_logits = model(x, edge_index, edge_weight)
x = self.conv2(x, edge_index, edge_weight)
x = F.relu(self.conv1(x, edge_index, edge_weight))
def forward(self, x, edge_index, edge_weight):
```

while edge weight = where data=None in In pytorch_geometric example:

hence, edge_weight is not necessary