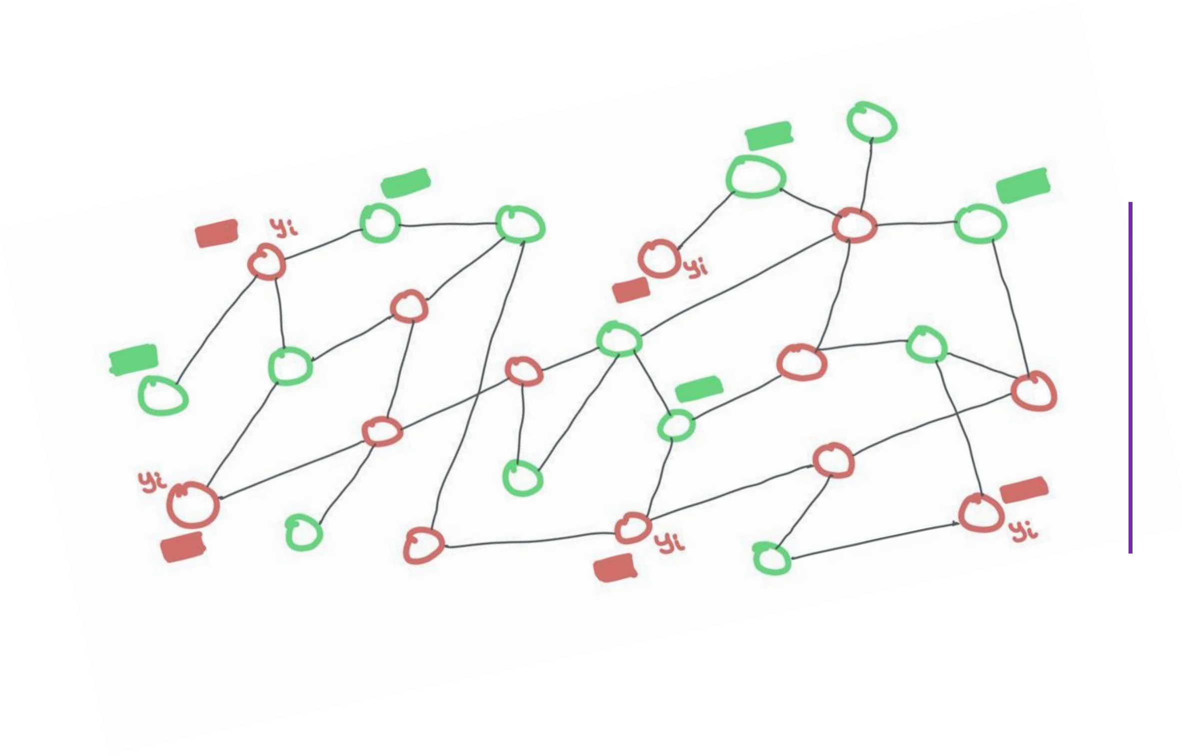


Node Classification in Citation Networks using Graph Neural Networks (GNNs)

Habib Heidari (Ryan Heida)
401651114

Project Description

Project title

Node Classification in Citation Networks using Graph Neural Networks (GNNs)

Objective

The main goal of this project is to apply Graph Neural Networks (GNNs) to a citation network to classify academic papers based on their subject areas. Citation networks are represented as graphs where:

- **Nodes**: Represent academic papers.
- **Edges**: Represent citation relationships between papers.

Each paper (node) is linked to others that cite it or are cited by it. The objective is to use GNNs to predict the subject category of each paper based on the citation network structure and paper features.

Dataset

Cora Dataset

The Cora dataset is commonly used for projects of this type and includes:

- **Nodes**: Papers.
- **Edges**: Citation links between papers.
- **Features**: A sparse bag-of-words representation for each paper.
- **Labels**: Subject areas (e.g., Machine Learning, Data Mining, Neural Networks, etc.).

The dataset can be sourced from libraries like PyTorch Geometric or TensorFlow Graphs.

Code

```python
import torch
import torch.nn.functional as F
from torch_geometric.nn import GCNConv
from torch_geometric.datasets import Planetoid
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE

# Loading dataset
dataset = Planetoid(root='data', name='Cora')
data = dataset[0]

# Defining the model
class GCN(torch.nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(GCN, self).__init__()
        self.conv1 = GCNConv(input_dim, hidden_dim)
        self.conv2 = GCNConv(hidden_dim, output_dim)

    def forward(self, data):
        x, edge_index = data.x, data.edge_index
```

Code

```python
# Defining the model
class GCN(torch.nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(GCN, self).__init__()
        self.conv1 = GCNConv(input_dim, hidden_dim)
        self.conv2 = GCNConv(hidden_dim, output_dim)

    def forward(self, data):
        x, edge_index = data.x, data.edge_index
        x = self.conv1(x, edge_index)
        x = F.relu(x)
        x = self.conv2(x, edge_index)
        return F.log_softmax(x, dim=1)

# Training and testing functions
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = GCN(input_dim=dataset.num_node_features, hidden_dim=16,
output_dim=dataset.num_classes).to(device)
data = data.to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.01, weight_decay=5e-4)

def train():
    model.train()
    optimizer.zero_grad()
    out = model(data)
    loss = F.nll_loss(out[data.train_mask], data.y[data.train_mask])
    loss.backward()
    optimizer.step()
    return loss.item()

def test():
    model.eval()
    with torch.no_grad():
```

Code

```
def train():
    model.train()
    optimizer.zero_grad()
    out = model(data)
    loss = F.nll_loss(out[data.train_mask], data.y[data.train_mask])
    loss.backward()
    optimizer.step()
    return loss.item()

def test():
    model.eval()
    with torch.no_grad():
        logits = model(data)
        test_mask = data.test_mask
        pred = logits[test_mask].max(1)[1]
        acc = accuracy_score(data.y[test_mask].cpu(), pred.cpu())
        prec = precision_score(data.y[test_mask].cpu(), pred.cpu(), average='weighted')
        rec = recall_score(data.y[test_mask].cpu(), pred.cpu(), average='weighted')
        f1 = f1_score(data.y[test_mask].cpu(), pred.cpu(), average='weighted')
    return acc, prec, rec, f1

# Function to visualize node embeddings using t-SNE or PCA
def visualize_embeddings(embeddings, labels, method='pca'):
    if method == 'pca':
        reducer = PCA(n_components=2)
    elif method == 'tsne':
        reducer = TSNE(n_components=2)
    else:
        raise ValueError("Method must be 'pca' or 'tsne'")

    reduced_embeddings = reducer.fit_transform(embeddings)
    plt.figure(figsize=(8, 8))
    scatter = plt.scatter(reduced_embeddings[:, 0], reduced_embeddings[:, 1], c=labels, cmap='viridis',
```

Code

```python
# Function to visualize node embeddings using t-SNE or PCA
def visualize_embeddings(embeddings, labels, method='pca'):
    if method == 'pca':
        reducer = PCA(n_components=2)
    elif method == 'tsne':
        reducer = TSNE(n_components=2)
    else:
        raise ValueError("Method must be 'pca' or 'tsne'")

    reduced_embeddings = reducer.fit_transform(embeddings)
    plt.figure(figsize=(8, 8))
    scatter = plt.scatter(reduced_embeddings[:, 0], reduced_embeddings[:, 1], c=labels, cmap='viridis',
alpha=0.7)
    plt.colorbar(scatter, label="Classes")
    plt.title(f'Node Embeddings Visualization using {method.upper()}')
    plt.xlabel('Component 1')
    plt.ylabel('Component 2')
    plt.show()

# Tracking loss and accuracy over epochs
losses = []
accuracies = []

# Training the model and tracking accuracy
for epoch in range(200):
    loss = train()
    losses.append(loss)

    if epoch % 10 == 0:
        acc, prec, rec, f1 = test()
        accuracies.append(acc)
        print(f'Epoch {epoch}, Loss: {loss:.4f}, Accuracy: {acc:.4f}')
```

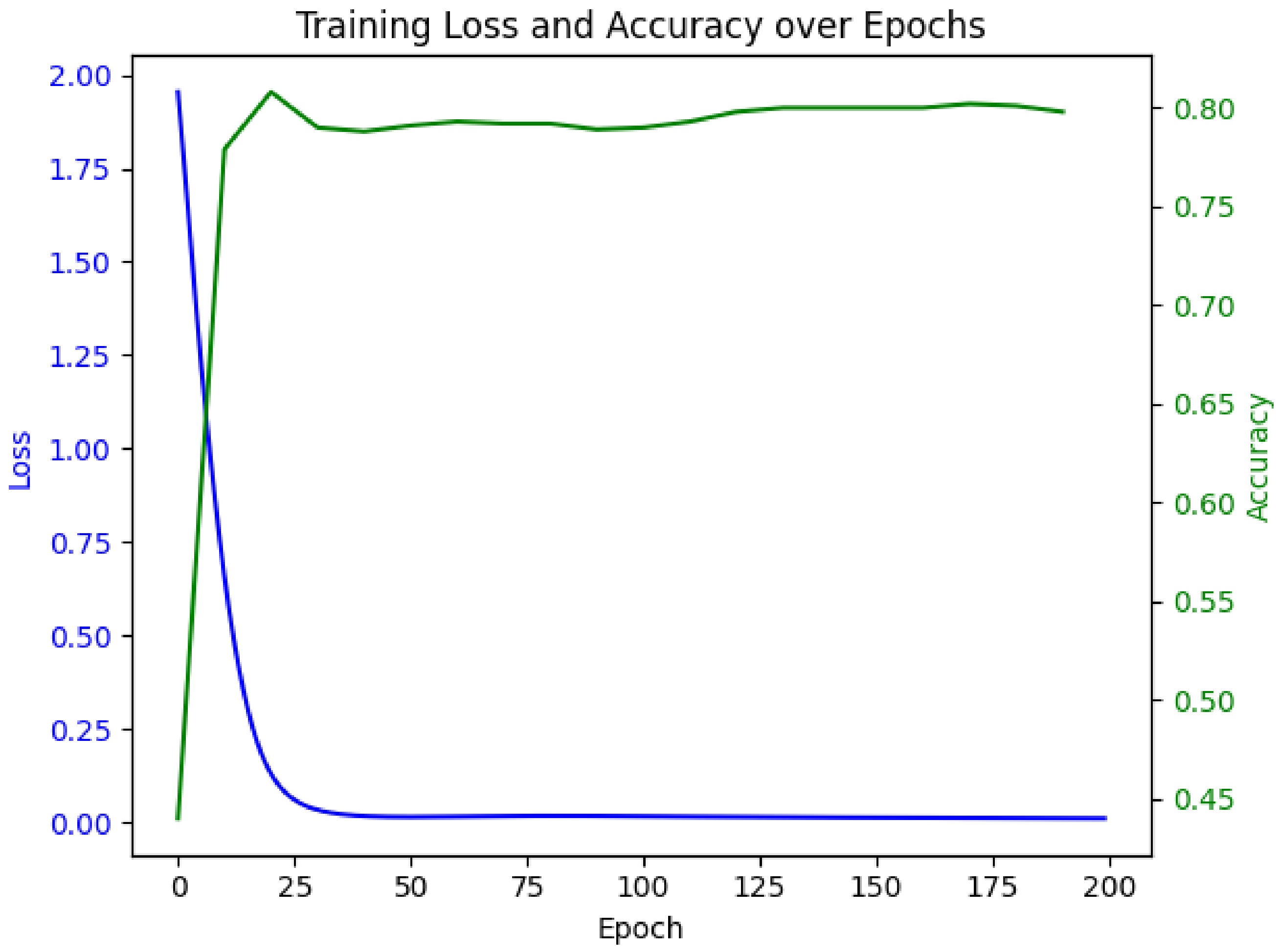
Code

```python
# Training the model and tracking accuracy
for epoch in range(200):
    loss = train()
    losses.append(loss)

    if epoch % 10 == 0:
        acc, prec, rec, f1 = test()
        accuracies.append(acc)
        print(f'Epoch {epoch}, Loss: {loss:.4f}, Accuracy: {acc:.4f}')

# Plot training loss and accuracy over epochs
fig, ax1 = plt.subplots()

ax1.plot(losses, label='Loss', color='blue')
ax1.set_xlabel('Epoch')
ax1.set_ylabel('Loss', color='blue')
ax1.tick_params(axis='y', labelcolor='blue')

ax2 = ax1.twinx()
ax2.plot(range(0, 200, 10), accuracies, label='Accuracy', color='green')
ax2.set_ylabel('Accuracy', color='green')
ax2.tick_params(axis='y', labelcolor='green')

plt.title('Training Loss and Accuracy over Epochs')
fig.tight_layout()
plt.show()

# Get the final embeddings from the model
model.eval()
with torch.no_grad():
    final_embeddings = model(data).cpu().numpy()

# Visualize embeddings using t-SNE or PCA
```

```
# Plot training loss and accuracy over epochs
fig, ax1 = plt.subplots()

ax1.plot(losses, label='Loss', color='blue')
ax1.set_xlabel('Epoch')
ax1.set_ylabel('Loss', color='blue')
ax1.tick_params(axis='y', labelcolor='blue')

ax2 = ax1.twinx()
ax2.plot(range(0, 200, 10), accuracies, label='Accuracy', color='green')
ax2.set_ylabel('Accuracy', color='green')
ax2.tick_params(axis='y', labelcolor='green')

plt.title('Training Loss and Accuracy over Epochs')
fig.tight_layout()
plt.show()

# Get the final embeddings from the model
model.eval()
with torch.no_grad():
    final_embeddings = model(data).cpu().numpy()

# Visualize embeddings using t-SNE or PCA
visualize_embeddings(final_embeddings, data.y.cpu(), method='tsne')
visualize_embeddings(final_embeddings, data.y.cpu(), method='pca')

# Evaluating the final model performance
acc, prec, rec, f1 = test()
print(f'Accuracy: {acc:.4f}, Precision: {prec:.4f}, Recall: {rec:.4f}, F1-Score: {f1:.4f}')
```

Code

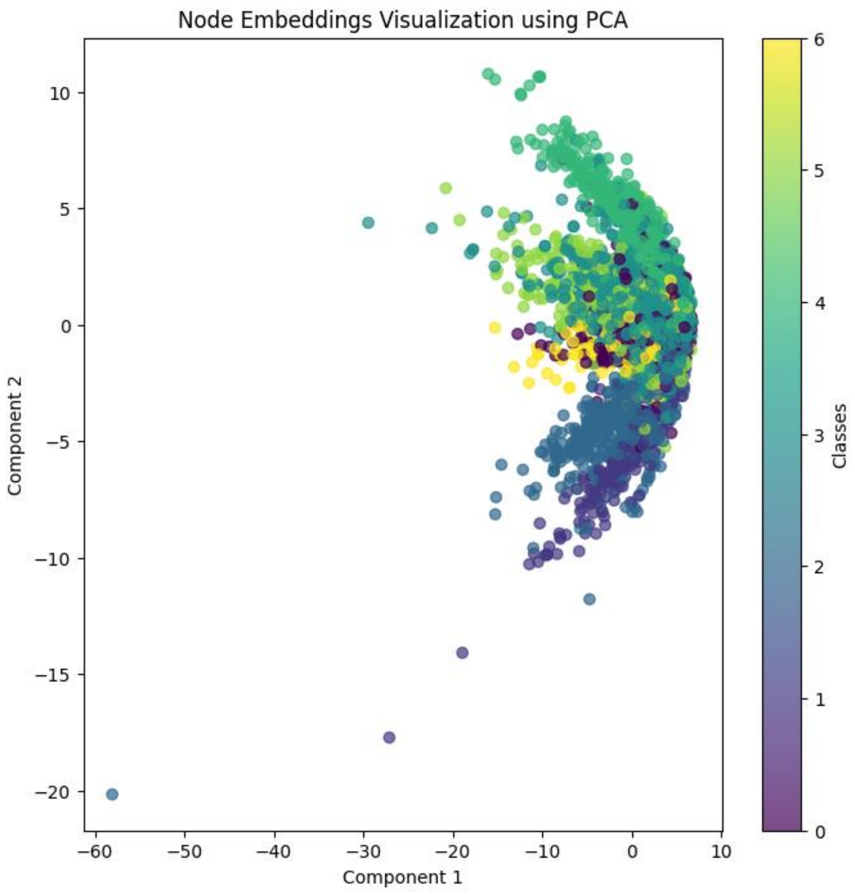
Loss / Accuracy & Statistics



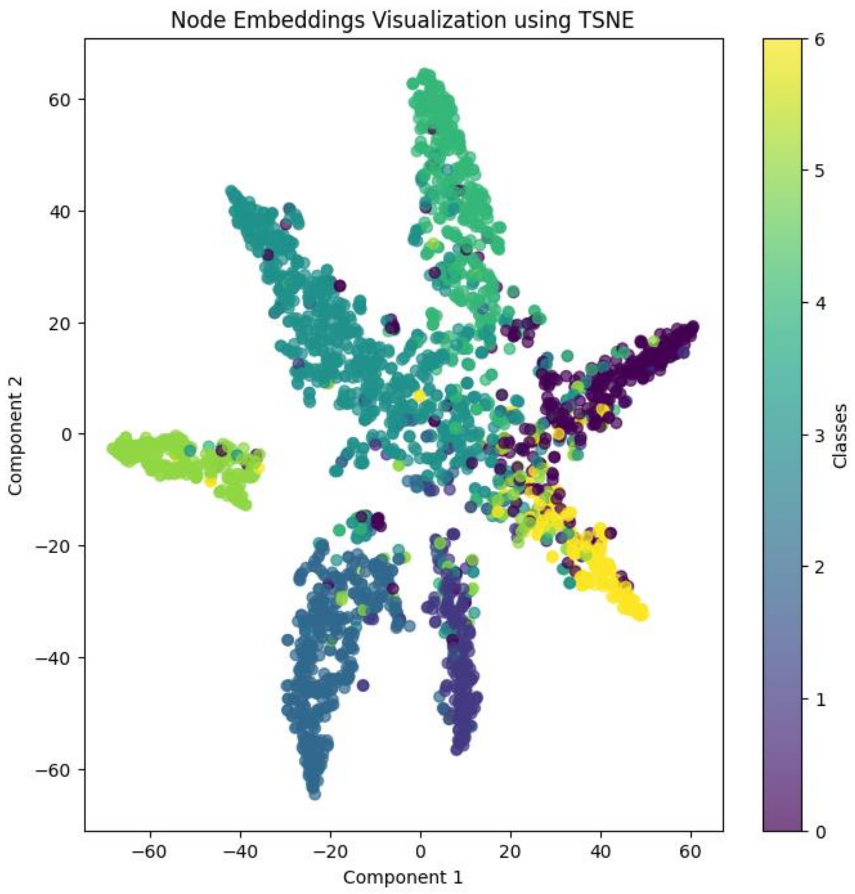
Final Result

Accuracy: 0.8020
Precision: 0.8120
Recall: 0.8020
F1-Score: 0.8030

Visualization



PCA



t-SNE

Extensions - GAT

Implementation

```
from torch_geometric.nn import GATConv

class GAT(torch.nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, heads=8):
        super(GAT, self).__init__()
        # First GAT layer (multi-head attention)
        self.conv1 = GATConv(input_dim, hidden_dim, heads=heads, dropout=0.6)
        # Second GAT layer (single attention head for output)
        self.conv2 = GATConv(hidden_dim * heads, output_dim, heads=1, concat=False, dropout=0.6)

    def forward(self, data):
        x, edge_index = data.x, data.edge_index
        x = F.elu(self.conv1(x, edge_index))
        x = self.conv2(x, edge_index)
        return F.log_softmax(x, dim=1)
```

Results

Accuracy: 0.7790
Precision: 0.8012
Recall: 0.7790
F1-Score: 0.7811

Extensions - Explore hyperparameters

Implementation

```
class GCN(torch.nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, num_layers=2):
        super(GCN, self).__init__()
        self.num_layers = num_layers
        self.conv1 = GCNConv(input_dim, hidden_dim)
        if num_layers > 1:
            self.conv2 = GCNConv(hidden_dim, output_dim)
        else:
            self.conv2 = None

    def forward(self, data):
        x, edge_index = data.x, data.edge_index
        x = self.conv1(x, edge_index)
        x = F.relu(x)
        if self.conv2 is not None:
            x = self.conv2(x, edge_index)
        return F.log_softmax(x, dim=1)

hyperparams = [
    {'hidden_dim': 16, 'num_layers': 2, 'learning_rate': 0.01},
    {'hidden_dim': 32, 'num_layers': 2, 'learning_rate': 0.01},
    {'hidden_dim': 16, 'num_layers': 1, 'learning_rate': 0.005},
    {'hidden_dim': 32, 'num_layers': 1, 'learning_rate': 0.005},
    {'hidden_dim': 64, 'num_layers': 2, 'learning_rate': 0.001}
]

for i, params in enumerate(hyperparams):
    print(f"\nExperiment {i + 1} with params: {params}")

    model = GCN(input_dim=dataset.num_node_features, hidden_dim=params['hidden_dim'],
                output_dim=dataset.num_classes, num_layers=params['num_layers']).to(device)
    optimizer = torch.optim.Adam(model.parameters(), lr=params['learning_rate'], weight_decay=5e-4)
```

Extensions - Explore hyperparameters

Results

- **Experiment 1** with params: {'hidden_dim': 16, 'num_layers': 2, 'learning_rate': 0.01}
  - **Accuracy**: 0.8050, **Precision**: 0.8156, **Recall**: 0.8050, **F1-Score**: 0.8066
- **Experiment 2** with params: {'hidden_dim': 32, 'num_layers': 2, 'learning_rate': 0.01}
  - **Accuracy**: 0.8140, **Precision**: 0.8242, **Recall**: 0.8140, **F1-Score**: 0.8155
- **Experiment 3** with params: {'hidden_dim': 16, 'num_layers': 1, 'learning_rate': 0.005}
  - **Accuracy**: 0.7270, **Precision**: 0.7484, **Recall**: 0.7270, **F1-Score**: 0.7283
- **Experiment 4** with params: {'hidden_dim': 32, 'num_layers': 1, 'learning_rate': 0.005}
  - **Accuracy**: 0.7330, **Precision**: 0.7524, **Recall**: 0.7330, **F1-Score**: 0.7344
- **Experiment 5** with params: {'hidden_dim': 64, 'num_layers': 2, 'learning_rate': 0.001}
  - **Accuracy**: 0.7970, **Precision**: 0.8108, **Recall**: 0.7970, **F1-Score**: 0.7987

Extensions - Test on other datasets

Implementation

```python
datasets = ['Cora', 'PubMed', 'CiteSeer']

for dataset_name in datasets:
    print(f"\nTesting on {dataset_name} dataset")
    dataset = Planetoid(root='../data', name=dataset_name)
```

Results

- **Cora dataset**
  - **Accuracy**: 0.8010, **Precision**: 0.8162, **Recall**: 0.8010, **F1-Score**: 0.8025
- **PubMed dataset**
  - **Accuracy**: 0.7910, **Precision**: 0.7946, **Recall**: 0.7910, **F1-Score**: 0.7903
- **CiteSeer dataset**
  - **Accuracy**: 0.6820, **Precision**: 0.7049, **Recall**: 0.6820, **F1-Score**: 0.6889

Extensions - Add Features

(Like Publication Year or Journal Impact Factor for Enhanced Modeling)

Implementation

```
num_nodes = data.num_nodes
publication_year = np.random.randint(2000, 2021, size=(num_nodes, 1))
impact_factor = np.random.uniform(0.1, 10.0, size=(num_nodes, 1))

additional_features = torch.tensor(np.hstack([publication_year, impact_factor]), dtype=torch.float)
data.x = torch.cat([data.x, additional_features.to(device)], dim=1)
```

Results

Accuracy: 0.7110
Precision: 0.7711
Recall: 0.7110
F1-Score: 0.7198