

I have a dataset of microservice communications source and destination and kpi metrics like error rate, throughput, and latency with timestamp. I labeled the dataset using the error rate feature as healthy if the rate is 1, degraded if the rate is less than 0.1 else it is an error. Now I want to build a graph based model maybe GNN or GCN to predict the status of communication between the microservices. I want the time stamp to be considered in the training so we have temporal graphs

💡 Building a Temporal Graph Neural Network for Microservice Communication Status Prediction

Based on your description, you're looking to build a graph-based model that can predict the status of communications between microservices, incorporating both the graph structure and temporal information. Here's a comprehensive approach:

Problem Formulation

You have:

- A dynamic graph where nodes represent microservices and edges represent communications
- Edge features including error rate, throughput, and latency
- Timestamped data that captures how these communications evolve over time
- Three-class labels (healthy, degraded, error) based on error rate thresholds

Solution Architecture

A Temporal Graph Neural Network (TGNN) would be ideal for this problem. Here are the key components:

1. Graph Representation

- **Nodes:** Each microservice is a node

- **Edges:** Communications between microservices (directed or undirected based on your needs)
- **Edge features:** [error_rate, throughput, latency] for each communication
- **Node features:** Optional - could include microservice metadata or aggregated metrics

2. Temporal Handling

Since you have timestamped data, you'll need to handle the temporal aspect. Two main approaches:

Option A: Discrete-time Temporal GNN

- Split your data into time slices (e.g., 5-minute intervals)
- Create a sequence of graph snapshots (G_1, G_2, \dots, G_t)
- Use a model like TGAT (Temporal Graph Attention Network) or DySAT

Option B: Continuous-time Temporal GNN

- Model events as a continuous-time process
- Use methods like TGNN or TGN (Temporal Graph Networks)

3. Model Architecture

Here's a sample architecture using PyTorch Geometric Temporal:

```
python
from torch_geometric_temporal.nn.recurrent import TGCN2

class TemporalGNN(torch.nn.Module):
    def __init__(self, node_features, edge_features, periods):
        super().__init__()
        # Temporal graph convolution layer
        self.tggnn = TGCN2(node_features, node_features, improved=True)

        # Edge prediction head
        self.edge_predictor = torch.nn.Sequential(
            torch.nn.Linear(2*node_features + edge_features, 64),
            torch.nn.ReLU(),
            torch.nn.Linear(64, 3)  # 3 classes
        )

    def forward(self, x, edge_index, edge_attr, edge_time):
```

Copy

```

# x: Node features
# edge_index: Graph connectivity
# edge_attr: Edge features
# edge_time: Temporal information

# Update node embeddings with temporal information
h = self.tggn(x, edge_index, edge_time)

# Get source and target node embeddings for each edge
src, dst = edge_index
h_src = h[src]
h_dst = h[dst]

# Concatenate for edge classification
edge_emb = torch.cat([h_src, h_dst, edge_attr], dim=1)
return self.edge_predictor(edge_emb)

```

4. Data Preparation

You'll need to create a temporal graph dataset. Here's a conceptual approach:

1. Sort all communications by timestamp
2. Create time windows (e.g., 5-minute intervals)
3. For each window, create a graph snapshot with:
 - Nodes: All microservices active in that window
 - Edges: Communications with their features
 - Labels: Your status labels (healthy, degraded, error)

5. Training Approach

- Use a sliding window approach for training
- For each time step t :
 - Use graph snapshots from $t-k$ to $t-1$ to predict status at t
- Loss function: Cross-entropy loss for multi-class classification
- Metrics: Accuracy, precision, recall, F1-score (especially important for imbalanced classes)

Implementation Considerations

1. **Graph Dynamics:** Microservices might appear/disappear over time - your model should handle this

2. **Class Imbalance:** Healthy communications likely dominate - consider weighted loss or oversampling
3. **Feature Scaling:** Normalize throughput and latency features
4. **Temporal Granularity:** Choose time windows that match your operational needs (e.g., 1 minute for real-time monitoring)
5. **Edge Direction:** Decide if communications are directional ($A \rightarrow B$) or bidirectional

Evaluation

- Temporal cross-validation: Train on earlier time periods, validate on later ones
- Compare against:
 - Static GNN (ignores temporal information)
 - Traditional time series models (ignores graph structure)
 - Baseline heuristic (e.g., "if error_rate > threshold → error")

Tools & Libraries

- **PyTorch Geometric Temporal:** For temporal GNN implementations
- **DGL:** Alternative library with temporal graph support
- **NetworkX:** For graph analysis and visualization
- **Pandas:** For handling time series data

Would you like me to elaborate on any specific aspect of this approach, or do you have particular constraints (e.g., real-time requirements) that might affect the architecture?

i used the error rate to lable the communication, can we use it as feature to train the model or we should eliminate it from features

Great question! This is an important consideration when building your model. Here's my analysis of whether to include the error rate as a feature:

Should You Use Error Rate as Both Label and Feature?

The Problem:

You've created labels based on error rate thresholds, so using the same error rate as a feature would create **data leakage** - the model could trivially learn to predict the label by just looking at the error rate value, without learning any meaningful patterns from other features or the graph structure.

Recommended Approaches:

1. Exclude Error Rate from Features (Safer Choice):

- **Pros:** Prevents any risk of leakage, forces model to learn from other signals
- **Cons:** Might discard useful information
- **Best for:** Production systems where you want pure behavioral prediction

2. Include Error Rate but with Time Delay:

- Use error rate from time $t-1$ to predict status at time t
- **Pros:** Realistic operational scenario (you know past error rates)
- **Cons:** More complex implementation

3. Two-Stage Approach:

- First model: Predicts error rate from other features
- Second stage: Apply your threshold rules to the predicted error rate
- **Pros:** More nuanced probability outputs
- **Cons:** More complex pipeline

4. Include Derived Error Rate Features:

- Instead of raw error rate, use:
 - Rolling averages
 - Difference from historical baseline
 - Rate-of-change
- **Pros:** Maintains temporal patterns without direct leakage
- **Cons:** Requires careful feature engineering

Implementation Example (Option 1 - Excluding Error Rate):

```
python
```

```
Copy
```

```
# When preparing your edge features:  
# Instead of:  
edge_features = [error_rate, throughput, latency] # Includes error rate - BAD
```

```
# Do:  
edge_features = [throughput, latency] # Excludes error rate - GOOD  
  
# Your labels remain based on error rate thresholds  
labels = apply_error_rate_thresholds(error_rates)
```

If You Choose Option 2 (Time-Delayed Features):

python

Copy

```
# For time step t:  
# Features use metrics from t-1  
edge_features_t = [throughput_t_minus_1, latency_t_minus_1]  
  
# Label uses error rate at t  
label_t = classify_error_rate(error_rate_t)
```

Additional Considerations:

1. **Throughput-Latency Relationship:** These often correlate with error conditions, so the model can still learn patterns:
 - High latency + low throughput → Might indicate problems
 - Spiky patterns → Might predict upcoming errors
2. **Graph Structure Signals:** Even without error rate, the model can learn:
 - Which service pairs often have problems
 - Cascading failure patterns
 - Temporal propagation of issues
3. **Evaluation:** If you exclude error rate, check if model performance is still good:
 - If accuracy remains high → Other features are predictive
 - If accuracy drops too much → May need to reconsider feature exclusion

Would you like me to elaborate on any particular approach or discuss how this would work with your specific graph architecture?

I will go with option 1 and eliminate the error rate. I want now to proceed tp the prediction of communication status using time windows graph of 1 minute and i have the communication ordered already. can you provide a complete code handling with evaluation



Temporal Graph Neural Network for Microservice Status Prediction

Here's a complete implementation for predicting communication status using 1-minute time windows, excluding the error rate from features as you requested.

1. Data Preparation

```
python Copy
import numpy as np
import pandas as pd
import torch
from torch_geometric.data import Data
from torch_geometric_temporal.signal import TemporalDataLoader, StaticGraphTempora
lSignal
from sklearn.metrics import classification_report, f1_score
from sklearn.preprocessing import StandardScaler

# Assuming your data is in a DataFrame with columns:
# ['timestamp', 'source', 'target', 'throughput', 'latency', 'error_rate', 'status']
# where status is already labeled (healthy=1, degraded=0.5, error=0)

# 1. Load and preprocess data
df = pd.read_csv('microservice_communications.csv')
df['timestamp'] = pd.to_datetime(df['timestamp'])

# Sort by timestamp
df = df.sort_values('timestamp')

# Create 1-minute time windows
df['time_window'] = df['timestamp'].dt.floor('1min')

# Exclude error_rate from features as per Option 1
features = ['throughput', 'latency']
target = 'status'

# Normalize features
scaler = StandardScaler()
df[features] = scaler.fit_transform(df[features])

# Get all unique microservices
all_nodes = sorted(list(set(df['source'].unique()).union(set(df['target'].unique()))))
node_map = {node: i for i, node in enumerate(all_nodes)}
num_nodes = len(all_nodes)
```

2. Create Temporal Graph Dataset

python

Copy

```
class MicroserviceCommunicationDataset:
    def __init__(self, df, node_map, features, target):
        self.df = df
        self.node_map = node_map
        self.features = features
        self.target = target
        self.time_windows = sorted(df['time_window'].unique())

    def __len__(self):
        return len(self.time_windows) - 1 # Predict next window

    def __getitem__(self, idx):
        # Use window t to predict window t+1
        current_window = self.time_windows[idx]
        next_window = self.time_windows[idx+1]

        # Get current window data (for node features)
        current_data = self.df[self.df['time_window'] == current_window]

        # Get next window data (for edge predictions)
        next_data = self.df[self.df['time_window'] == next_window]

        # Create edge index and features for prediction
        edge_index = []
        edge_attrs = []
        edge_labels = []

        for _, row in next_data.iterrows():
            src = self.node_map[row['source']]
            tgt = self.node_map[row['target']]
            edge_index.append([src, tgt])
            edge_attrs.append(row[self.features].values)
            edge_labels.append(row[self.target])

        if not edge_index: # Handle empty windows
            edge_index = torch.empty((2, 0), dtype=torch.long)
            edge_attrs = torch.empty((0, len(self.features)), dtype=torch.float)
            edge_labels = torch.empty((0,), dtype=torch.long)
        else:
            edge_index = torch.tensor(edge_index, dtype=torch.long).t().contiguous()
            edge_attrs = torch.tensor(edge_attrs, dtype=torch.float)
            edge_labels = torch.tensor(edge_labels, dtype=torch.long)

        # Create graph data object
        data = Data(
```

```

        x=torch.randn((num_nodes, 16)), # Random initial node features
        edge_index=edge_index,
        edge_attr=edge_attrs,
        y=edge_labels
    )

    return data

def get_edges(self, window):
    """Helper function to get edges for a specific window"""
    window_data = self.df[self.df['time_window'] == window]
    edge_index = []
    for _, row in window_data.iterrows():
        src = self.node_map[row['source']]
        tgt = self.node_map[row['target']]
        edge_index.append([src, tgt])
    return torch.tensor(edge_index, dtype=torch.long).t().contiguous() if edge_index
else None

# Create dataset
dataset = MicroserviceCommunicationDataset(df, node_map, features, target)

# Split into train/val/test (70%/15%/15%)
train_size = int(0.7 * len(dataset))
val_size = int(0.15 * len(dataset))
test_size = len(dataset) - train_size - val_size

train_dataset = [dataset[i] for i in range(train_size)]
val_dataset = [dataset[i] for i in range(train_size, train_size + val_size)]
test_dataset = [dataset[i] for i in range(train_size + val_size, len(dataset))]

```

3. Model Architecture

python

Copy

```

import torch.nn as nn
import torch.nn.functional as F
from torch_geometric.nn import GCNConv, TemporalGNN
from torch_geometric_temporal.nn.recurrent import TGCN2

class TemporalGNNModel(nn.Module):
    def __init__(self, node_features, edge_features, hidden_dim, num_classes):
        super().__init__()
        self.tggnn = TGCN2(node_features, hidden_dim)
        self.edge_predictor = nn.Sequential(
            nn.Linear(2 * hidden_dim + edge_features, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, num_classes)
    )

```

```

def forward(self, data):
    # Update node embeddings
    h = self.tggn(data.x, data.edge_index)

    # Get source and target node embeddings for each edge
    src, dst = data.edge_index
    h_src = h[src]
    h_dst = h[dst]

    # Concatenate node embeddings with edge features
    edge_emb = torch.cat([h_src, h_dst, data.edge_attr], dim=1)

    # Predict edge status
    return self.edge_predictor(edge_emb)

# Initialize model
model = TemporalGNNModel(
    node_features=16, # Match initial random features
    edge_features=len(features),
    hidden_dim=64,
    num_classes=3 # healthy, degraded, error
)

```

4. Training Loop

```

python                                         Copy
from torch_geometric_temporal.signal import temporal_signal_split
from torch_geometric_temporal.nn.recurrent import GCLSTM

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = model.to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
criterion = nn.CrossEntropyLoss()

def train():
    model.train()
    total_loss = 0
    for data in train_dataset:
        data = data.to(device)
        optimizer.zero_grad()
        out = model(data)
        loss = criterion(out, data.y)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    return total_loss / len(train_dataset)

def evaluate(dataset):

```

```

model.eval()
preds, truths = [], []
with torch.no_grad():
    for data in dataset:
        data = data.to(device)
        out = model(data)
        pred = out.argmax(dim=1)
        preds.append(pred.cpu())
        truths.append(data.y.cpu())

preds = torch.cat(preds).numpy()
truths = torch.cat(truths).numpy()

print(classification_report(truths, preds, target_names=['healthy', 'degraded', 'error']))
print("Macro F1-score:", f1_score(truths, preds, average='macro'))

return f1_score(truths, preds, average='macro')

# Training loop
best_val_f1 = 0
for epoch in range(1, 101):
    loss = train()
    val_f1 = evaluate(val_dataset)

    if val_f1 > best_val_f1:
        best_val_f1 = val_f1
        torch.save(model.state_dict(), 'best_model.pt')

    print(f'Epoch: {epoch:03d}, Loss: {loss:.4f}, Val F1: {val_f1:.4f}')

# Load best model and test
model.load_state_dict(torch.load('best_model.pt'))
print("Final Test Results:")
test_f1 = evaluate(test_dataset)

```

5. Evaluation and Visualization

python

Copy

```

import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

def plot_confusion_matrix(dataset):
    model.eval()
    preds, truths = [], []
    with torch.no_grad():
        for data in dataset:
            data = data.to(device)
            out = model(data)
            pred = out.argmax(dim=1)
            preds.append(pred.cpu())
            truths.append(data.y.cpu())

    cm = confusion_matrix(truths, preds)
    disp = ConfusionMatrixDisplay(cm, display_labels=['Healthy', 'Degrade'])
    disp.plot()
    plt.show()

```

```

        out = model(data)
        pred = out.argmax(dim=1)
        preds.append(pred.cpu())
        truths.append(data.y.cpu())

    preds = torch.cat(preds).numpy()
    truths = torch.cat(truths).numpy()

    cm = confusion_matrix(truths, preds, normalize='true')
    disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=['healthy', 'degraded', 'error'])
    disp.plot(cmap=plt.cm.Blues)
    plt.title('Normalized Confusion Matrix')
    plt.show()

# Plot confusion matrices
print("Validation Confusion Matrix:")
plot_confusion_matrix(val_dataset)

print("Test Confusion Matrix:")
plot_confusion_matrix(test_dataset)

# Plot training curves (if you tracked metrics during training)
# plt.plot(train_losses, label='Training Loss')
# plt.plot(val_f1s, label='Validation F1')
# plt.legend()
# plt.show()

```

Key Considerations:

1. **Temporal Dependency:** The model uses previous time windows to predict the next window's status.
2. **Class Imbalance Handling:** The cross-entropy loss naturally handles some imbalance, but you might want to add class weights if the imbalance is severe.
3. **Edge Cases:** The code handles empty time windows gracefully.
4. **Node Features:** Currently using random initial features. You could enhance this with:
 - Historical aggregated metrics
 - Microservice metadata
 - Learned embeddings
5. **Temporal Aggregation:** For better performance, consider:
 - Longer historical context (e.g., past 5 windows)
 - Attention mechanisms over time

Would you like me to modify any part of this implementation to better match your specific requirements or data structure?

I got this error

It looks like you encountered an error while running the code. To help you effectively, I'll need a few more details:

1. **The exact error message** (please share the full traceback)
2. **Which part of the code were you executing** when the error occurred?
3. **A sample of your data structure** (if different from what I assumed)

However, I can anticipate some common issues and provide solutions:

Common Issues and Fixes:

1. **KeyError: 'status'**:

```
python                                         Copy  
  
# If your label column has a different name, update:  
target = 'your_label_column_name' # instead of 'status'
```

2. **Shape Mismatch Errors**:

```
python                                         Copy  
  
# Ensure feature dimensions match:  
print(f"
```

ModuleNotFoundError: No module named 'torch_geometric_temporal'

This error occurs because you haven't installed the `torch_geometric_temporal` package, which is needed for temporal graph neural networks. Let's fix this and ensure all dependencies are properly installed.

Installation Guide

1. First install PyTorch (choose the appropriate command for your system):

```
bash
```

```
Copy
```

```
# For CUDA 11.3
pip install torch torchvision torchaudio --extra-index-url https://download.pytorch.org/whl/cu113

# For CPU-only
pip install torch torchvision torchaudio
```

2. Install PyTorch Geometric and its temporal extension:

```
bash
```

```
Copy
```

```
# Install dependencies first
pip install pyg-lib torch-scatter torch-sparse torch-cluster torch-spline-conv -f https://data.pyg.org/whl/torch-1.13.0+cu113.html

# Then install the main packages
pip install torch-geometric
pip install torch-geometric-temporal
```

Alternative Solution

If you're having trouble with the above installation, try this simplified approach:

```
bash
```

```
Copy
```

```
# Create new conda environment (recommended)
conda create -n tnat python=3.9
conda activate tnat

# Install basic packages
pip install torch torchvision torchaudio

# Install PyG and temporal extension
pip install torch-geometric
pip install torch-geometric-temporal
```

Verification

After installation, verify it works by running:

```
python
```

```
Copy
```

```
import torch_geometric
import torch_geometric_temporal
```

```
print(torch_geometric.__version__)
print(torch_geometric_temporal.__version__)
```

If You Still Get Errors

1. Check your PyTorch version:

```
python
import torch
print(torch.__version__)
```

Copy

2. Install specific versions that are known to work together:

```
bash
pip install torch==1.12.0
pip install torch-geometric==2.0.4
pip install torch-geometric-temporal==0.54.0
```

Copy

Would you like me to:

1. Provide a CPU-only installation guide
2. Suggest alternative temporal GNN approaches without this package
3. Help debug other specific errors you're encountering?

The installation can be tricky due to CUDA/cuDNN compatibility issues - sharing your exact environment details would help me give more tailored advice.

my torch version is 1.12.0+cu102

 Edit

 Since you have **Torch 1.12.0+cu102**, we need to install compatible versions of `torch-geometric` and `torch-geometric-temporal`. Here's the correct installation:

Step-by-Step Installation

```
bash
# 1. First uninstall any existing incompatible versions
pip uninstall torch-geometric torch-geometric-temporal torch-scatter torch-sparse torch-cluster torch-spline-conv -y

# 2. Install PyTorch Geometric dependencies (FOR CUDA 10.2)
pip install torch-scatter torch-sparse torch-cluster torch-spline-conv -f https://d
```

Copy

```
ata.pyg.org/wheel/torch-1.12.0+cu102.html

# 3. Install main packages
pip install torch-geometric==2.0.4
pip install torch-geometric-temporal==0.54.0
```

Verification Test

```
python Copy

import torch
print(torch.__version__) # Should show 1.12.0+cu102

import torch_geometric
print(torch_geometric.__version__) # Should show 2.0.4

import torch_geometric_temporal
print(torch_geometric_temporal.__version__) # Should show 0.54.0
```

If You Still Get Errors

Try these alternatives:

Option 1: CPU-Only Installation

```
bash Copy

pip install torch==1.12.0+cpu -f https://download.pytorch.org/wheel/torch_stable.html
pip install torch-geometric==2.0.4 torch-geometric-temporal==0.54.0 --no-index
```

Option 2: Newer Version Stack

```
bash Copy

# For newer versions (requires CUDA 11.3)
pip install torch==1.13.0+cu113 --extra-index-url https://download.pytorch.org/wheel/cu113
pip install torch-geometric torch-geometric-temporal
```

Would you like me to:

1. Provide a simplified model that doesn't need `torch-geometric-temporal`
2. Help debug other specific errors in your code
3. Suggest alternative temporal graph approaches?

