draft

April 14, 2024

CS598: Deep Learning for Healthcare Final Project

Team #164

Team member: Zexi Yan, Stanley Wen, and Bo Zhang

Corresponding emails: zexiyan2@illinois.edu, keyanw2@illinois.edu, zhang375@illinois.edu

Github Repo: https://github.com/ericyan3000/CS598-DLH-Chet.git

1 Introduction

This is an introduction to your report, you should edit this text/mardown section to compose. In this text/markdown, you should introduce:

1.0.1 Background of the problem

The primary challenge addressed by the paper is the limitation of existing health event prediction models which consider diagnoses as independent entities, neglecting the clinical relationships among diseases. This oversight hinders the ability to effectively utilize combinational disease information and understand the dynamic nature of disease development over time. This leads to the two problems the paper is trying to address. - Disease combinations in medical practice form a global graph structure that reveals hidden patterns among diseases, with individual patient visits represented as local subgraphs. Despite the potential to predict future health events by analyzing these structures, common deep learning models like GRAM [2], Timeline [3], and G-BERT [4] do not utilize this graph structure for health event predictions. - The progression of a disease in a patient is dynamic, as evidenced by changing diagnosis priorities and the emergence of new diagnoses in EHR datasets like MIMIC-III [5]. This dynamic nature, where diseases evolve and impact patients differently over time, suggests the need for a model that can dynamically represent disease development and learn the transition from potential to actual diagnoses. ### Paper explanation The paper titled "Context-aware Health Event Prediction via Transition Functions on Dynamic Disease Graphs" [1] introduces a novel framework for improving health event predictions by incorporating dynamic disease relationships within EHR data. The authors propose a sophisticated model that constructs and utilizes dynamic disease graphs to represent the evolving relationships between different diagnoses as patients continue to visit healthcare facilities.

The innovation of the method lies in its ability to dynamically adjust disease representations and interactions based on a patient's history and current health state. This is achieved through the use of global disease co-occurrence graphs and patient-specific subgraphs, which adapt based on new health information. The model uses transition functions to model the changes in disease relevance and connections, reflecting the natural progression and regression of disease states over time.

In terms of effectiveness, the proposed method has shown to outperform existing models significantly, as demonstrated through rigorous testing on real-world EHR datasets. The results indicated improvements in prediction accuracy for various health events, showcasing the model's capability to handle the complex dynamics of disease progression.

2 Scope of Reproducibility:

1. Chet can outperform existing state-of-the-art health event prediction models by effectively utilizing the dynamic and combinational nature of disease information. Specifically, we will compare Chet with the following model:
   - CNN-based model: Deepr [2].
2. The introduction of transition functions and dynamic graph learning will provide a more nuanced understanding of disease progression, leading to more accurate health event predictions. This will be tested by the Ablations described below.
   - Evaluate the model's performance without the transition functions on the prediction accuracy to understand their contribution to the model's performance.

3 Methodology

3.0.1 Data

The same preprocessed data will be used to feed each model and for the same task of generating predictions heart failure events.

3.0.2 Deepr Model (Control)

The Deepr model applies convolutional neural networks to sequences of diagnoses, treating each visit as a series of inputs to capture temporal and contextual patterns within patient data. This model serves as a baseline for comparison with more complex architectures.

3.0.3 Full Chet Model

The Chet model integrates dynamic disease graphs with a graph neural network (GNN) to capture both the global disease relationships across all patients and local changes specific to individual visits. It employs transition functions to model the temporal evolution of diseases, providing a comprehensive framework for predicting future health events.

3.0.4 Chet Model without Transition Functions (Chet-TF)

This variant of the Chet model retains the dynamic graph learning component but omits the transition functions. It focuses on the spatial relationships among diseases using GNN outputs directly for prediction, thus simplifying the approach by not modeling the temporal progression of diseases explicitly.

3.1 Data

- Source of the data: MIMIC III data to be used. Since it's not available yet, a demo data of MIMIC III is used instead.
  - https://physionet.org/content/mimiciii-demo/1.4/

[1]:
```
import os
import _pickle as pickle

from preprocess import save_sparse, save_data
from preprocess.parse_csv import Mimic3Parser
from preprocess.encode import encode_code
from preprocess.build_dataset import split_patients, build_code_xy,␣
 ↪build_heart_failure_y
from preprocess.auxiliary import generate_code_code_adjacent,␣
 ↪generate_neighbors, normalize_adj, divide_middle, generate_code_levels,␣
 ↪show_heart_failure_stat

# Configuration for dataset processing specific to the Mimic3 dataset
conf = {'mimic3': {
        'parser': Mimic3Parser,
        'train_num': 70,
        'test_num': 20,
        'threshold': 0.01
    }
}

# Flag to determine if data should be loaded from previously saved files
from_saved = True
data_path = 'data'
dataset = 'mimic3'
dataset_path = os.path.join(data_path, dataset)
raw_path = os.path.join(dataset_path, 'raw')

# Check if raw data directory exists, create if not and prompt for data␣
 ↪placement
if not os.path.exists(raw_path):
    os.makedirs(raw_path)
    print('please put the CSV files in `data/%s/raw`' % dataset)
    exit()

# Path for storing parsed data
parsed_path = os.path.join(dataset_path, 'parsed')

# Load or parse data depending on `from_saved` flag
if from_saved:
    # Load previously saved parsed data
    patient_admission = pickle.load(open(os.path.join(parsed_path,␣
 ↪'patient_admission.pkl'), 'rb'))
    admission_codes = pickle.load(open(os.path.join(parsed_path,␣
 ↪'admission_codes.pkl'), 'rb'))
else:
    # Parse new data from raw files
```

```
    parser = conf[dataset]['parser'](raw_path)
    sample_num = conf[dataset].get('sample_num', None)
    patient_admission, admission_codes = parser.parse(sample_num)
    print('saving parsed data ...')
    if not os.path.exists(parsed_path):
        os.makedirs(parsed_path)
    pickle.dump(patient_admission, open(os.path.join(parsed_path,␣
 ↪'patient_admission.pkl'), 'wb'))
    pickle.dump(admission_codes, open(os.path.join(parsed_path,␣
 ↪'admission_codes.pkl'), 'wb'))

# Calculate various statistics from the patient admissions data
patient_num = len(patient_admission)
max_admission_num = max([len(admissions) for admissions in patient_admission.
 ↪values()])
avg_admission_num = sum([len(admissions) for admissions in patient_admission.
 ↪values()]) / patient_num
max_visit_code_num = max([len(codes) for codes in admission_codes.values()])
avg_visit_code_num = sum([len(codes) for codes in admission_codes.values()]) /␣
 ↪len(admission_codes)
print('patient num: %d' % patient_num)
print('max admission num: %d' % max_admission_num)
print('mean admission num: %.2f' % avg_admission_num)
print('max code num in an admission: %d' % max_visit_code_num)
print('mean code num in an admission: %.2f' % avg_visit_code_num)

# Encode diagnosis codes and generate a code map
print('encoding code ...')
admission_codes_encoded, code_map = encode_code(patient_admission,␣
 ↪admission_codes)
code_num = len(code_map)
print('There are %d codes' % code_num)

# Generate levels for each code and save
code_levels = generate_code_levels(data_path, code_map)
pickle.dump({
    'code_levels': code_levels,
}, open(os.path.join(parsed_path, 'code_levels.pkl'), 'wb'))

# Split patients into training, validation, and test sets
train_pids, valid_pids, test_pids = split_patients(
    patient_admission=patient_admission,
    admission_codes=admission_codes,
    code_map=code_map,
    train_num=conf[dataset]['train_num'],
    test_num=conf[dataset]['test_num']
```

```
)
print('There are %d train, %d valid, %d test samples' % (len(train_pids),␣
 ↪len(valid_pids), len(test_pids)))
code_adj = generate_code_code_adjacent(pids=train_pids,␣
 ↪patient_admission=patient_admission,
                                        ␣
 ↪admission_codes_encoded=admission_codes_encoded,
                                         code_num=code_num,␣
 ↪threshold=conf[dataset]['threshold'])

# Additional data processing for training, validation, and test sets
common_args = [patient_admission, admission_codes_encoded, max_admission_num,␣
 ↪code_num]
print('building train codes features and labels ...')
(train_code_x, train_codes_y, train_visit_lens) = build_code_xy(train_pids,␣
 ↪*common_args)
print('building valid codes features and labels ...')
(valid_code_x, valid_codes_y, valid_visit_lens) = build_code_xy(valid_pids,␣
 ↪*common_args)
print('building test codes features and labels ...')
(test_code_x, test_codes_y, test_visit_lens) = build_code_xy(test_pids,␣
 ↪*common_args)

print('generating train neighbors ...')
train_neighbors = generate_neighbors(train_code_x, train_visit_lens, code_adj)
print('generating valid neighbors ...')
valid_neighbors = generate_neighbors(valid_code_x, valid_visit_lens, code_adj)
print('generating test neighbors ...')
test_neighbors = generate_neighbors(test_code_x, test_visit_lens, code_adj)

print('generating train middles ...')
train_divided = divide_middle(train_code_x, train_neighbors, train_visit_lens)
print('generating valid middles ...')
valid_divided = divide_middle(valid_code_x, valid_neighbors, valid_visit_lens)
print('generating test middles ...')
test_divided = divide_middle(test_code_x, test_neighbors, test_visit_lens)

print('building train heart failure labels ...')
train_hf_y = build_heart_failure_y('428', train_codes_y, code_map)
print('building valid heart failure labels ...')
valid_hf_y = build_heart_failure_y('428', valid_codes_y, code_map)
print('building test heart failure labels ...')
test_hf_y = build_heart_failure_y('428', test_codes_y, code_map)

# Save all processed data in appropriate paths
encoded_path = os.path.join(dataset_path, 'encoded')
```

```
if not os.path.exists(encoded_path):
    os.makedirs(encoded_path)
print('saving encoded data ...')
pickle.dump(patient_admission, open(os.path.join(encoded_path,␣
 ↪'patient_admission.pkl'), 'wb'))
pickle.dump(admission_codes_encoded, open(os.path.join(encoded_path,␣
 ↪'codes_encoded.pkl'), 'wb'))
pickle.dump(code_map, open(os.path.join(encoded_path, 'code_map.pkl'), 'wb'))
pickle.dump({
    'train_pids': train_pids,
    'valid_pids': valid_pids,
    'test_pids': test_pids
}, open(os.path.join(encoded_path, 'pids.pkl'), 'wb'))

print('saving standard data ...')
standard_path = os.path.join(dataset_path, 'standard')
train_path = os.path.join(standard_path, 'train')
valid_path = os.path.join(standard_path, 'valid')
test_path = os.path.join(standard_path, 'test')
if not os.path.exists(standard_path):
    os.makedirs(standard_path)
if not os.path.exists(train_path):
    os.makedirs(train_path)
    os.makedirs(valid_path)
    os.makedirs(test_path)

print('\tsaving training data')
save_data(train_path, train_code_x, train_visit_lens, train_codes_y,␣
 ↪train_hf_y, train_divided, train_neighbors)
print('\tsaving valid data')
save_data(valid_path, valid_code_x, valid_visit_lens, valid_codes_y,␣
 ↪valid_hf_y, valid_divided, valid_neighbors)
print('\tsaving test data')
save_data(test_path, test_code_x, test_visit_lens, test_codes_y, test_hf_y,␣
 ↪test_divided, test_neighbors)

code_adj = normalize_adj(code_adj)
save_sparse(os.path.join(standard_path, 'code_adj'), code_adj)

# Show statistics for heart failure

show_heart_failure_stat('428', train_codes_y, code_map)
show_heart_failure_stat('428', valid_codes_y, code_map)
show_heart_failure_stat('428', test_codes_y, code_map)
```

```
patient num: 100
max admission num: 300
```

```
mean admission num: 25.80
max code num in an admission: 740
mean code num in an admission: 273.02
encoding code …
There are 581 codes
generating code levels …
train_num: 70, test_num: 20
remaining_pids: 100
There are 70 train, 10 valid, 20 test samples
generating code code adjacent matrix …
        70 / 70
building train codes features and labels …
        70 / 70
building valid codes features and labels …
        10 / 10
building test codes features and labels …
        20 / 20
generating train neighbors …
        70 / 70
generating valid neighbors …
        10 / 10
generating test neighbors …
        20 / 20
generating train middles …
        70 / 70
generating valid middles …
        10 / 10
generating test middles …
        20 / 20
building train heart failure labels …
building valid heart failure labels …
building test heart failure labels …
saving encoded data …
saving standard data …
        saving training data
        saving valid data
        saving test data
24 patients have heart failure out of 70 patients
3 patients have heart failure out of 10 patients
7 patients have heart failure out of 20 patients
```

3.2 Model

The model includes the model definitation which usually is a class, model training, and other necessary parts. * Model architecture: layer number/size/type, activation function, etc * Training objectives: loss function, optimizer, weight of each loss term, etc * Others: whether the model is pretrained, Monte Carlo simulation for uncertainty analysis, etc * The code of model should have classes of the model, functions of model training, model validation, etc. * If your model training

is done outside of this notebook, please upload the trained model here and develop a function to load and test it.

3.2.1 Full Chet Model

- **SingleHeadAttentionLayer**: Implements a typical single-head attention mechanism, commonly found in architectures like the Transformer model. It uses linear transformations for the queries, keys, and values, computes scaled dot-product attention scores, and then computes a weighted sum of the values based on these scores.
- **DotProductAttention**: A simpler form of attention where a context vector learns to identify relevant features from the input data. It projects transformed inputs onto this context vector, calculates attention scores, and then weights the input data according to these scores.

[2]:
```
import math
import torch
from torch import nn

# Define a SingleHeadAttentionLayer class that extends nn.Module.
class SingleHeadAttentionLayer(nn.Module):
    def __init__(self, query_size, key_size, value_size, attention_size):
        super().__init__()  # Initialize the superclass (nn.Module)
        self.attention_size = attention_size  # Store the attention size

        # Define Linear transformations for query, key, and value vectors
        self.dense_q = nn.Linear(query_size, attention_size)  # Transforms␣
 ↪input query to the attention space
        self.dense_k = nn.Linear(key_size, attention_size)    # Transforms␣
 ↪input key to the attention space
        self.dense_v = nn.Linear(value_size, value_size)      # Transforms␣
 ↪input value, no change in dimension

    def forward(self, q, k, v):
        # Apply linear transformations
        query = self.dense_q(q)  # Transform query vector
        key = self.dense_k(k)    # Transform key vector
        value = self.dense_v(v)  # Transform value vector

        # Compute the attention scores
        # Scaled dot product attention mechanism
        g = torch.div(torch.matmul(query, key.T), math.sqrt(self.
 ↪attention_size))

        # Apply softmax to get attention weights
        score = torch.softmax(g, dim=-1)

        # Compute the weighted sum of values based on the attention scores
        output = torch.sum(torch.unsqueeze(score, dim=-1) * value, dim=-2)
```

```
        return output

# Define a DotProductAttention class that also extends nn.Module.
class DotProductAttention(nn.Module):
    def __init__(self, value_size, attention_size):
        super().__init__()  # Initialize the superclass (nn.Module)
        self.attention_size = attention_size  # Store the attention size
        # Initialize a context vector as a learnable parameter
        self.context = nn.Parameter(data=nn.init.xavier_uniform_(torch.
 ↪empty(attention_size, 1)))
        self.dense = nn.Linear(value_size, attention_size)  # Transforms input␣
 ↪value to the attention space

    def forward(self, x):
        # Transform input x to attention space
        t = self.dense(x)

        # Compute unnormalized attention scores by projecting 't' onto 'context'
        vu = torch.matmul(t, self.context).squeeze()

        # Apply softmax to get normalized attention weights
        score = torch.softmax(vu, dim=-1)

        # Compute the weighted sum of the original inputs based on the␣
 ↪attention weights
        output = torch.sum(x * torch.unsqueeze(score, dim=-1), dim=-2)
        return output
```

EmbeddingLayer

- Purpose: The EmbeddingLayer is responsible for transforming discrete code identifiers into dense vector representations. This is a fundamental step in many neural network models that deal with categorical data, enabling the model to capture and leverage the relationships and patterns inherent in the data more effectively.
- Functionality: Three Types of Embeddings: It initializes three types of embeddings for the codes: center (c_embeddings), neighbor (n_embeddings), and a general use embedding (u_embeddings). Each of these embeddings serves different roles in the graph-based computations that follow, potentially representing different aspects or features of the data.
- Parameter Initialization: The embeddings are initialized using Xavier uniform distribution, which is a common practice for initializing weights in neural networks in a way that aims to maintain the variance of activations across layers.

[3]:
```
import torch
from torch import nn

# EmbeddingLayer: Handles the embedding of codes into vectors.
class EmbeddingLayer(nn.Module):
```

```
    def __init__(self, code_num, code_size, graph_size):
        super().__init__()
        # Number of unique codes
        self.code_num = code_num
        # Embedding parameters initialized using Xavier uniform distribution
        # c_embeddings for center node embeddings
        self.c_embeddings = nn.Parameter(data=nn.init.xavier_uniform_(torch.
↪empty(code_num, code_size)))
        # n_embeddings for neighbor node embeddings
        self.n_embeddings = nn.Parameter(data=nn.init.xavier_uniform_(torch.
↪empty(code_num, code_size)))
        # u_embeddings for other uses, potentially for graph-level embeddings
        self.u_embeddings = nn.Parameter(data=nn.init.xavier_uniform_(torch.
↪empty(code_num, graph_size)))

    def forward(self):
        # Return all embeddings as outputs
        return self.c_embeddings, self.n_embeddings, self.u_embeddings
```

GraphLayer

- Purpose: The GraphLayer utilizes the embeddings provided by the EmbeddingLayer to perform computations reflecting the relationships and interactions encoded in an adjacency matrix. This layer is crucial for models that incorporate graph theory to process data structured as graphs (e.g., social networks, molecule structures).
- Functionality:
  - Embedding Interaction: It computes new embeddings based on interactions between center and neighbor nodes using the adjacency matrix. This involves matrix multiplication operations that simulate the propagation of information through the graph.
  - Transformation and Non-linearity: After computing the initial interactions, the embeddings are transformed through a linear layer and passed through a non-linear activation function (LeakyReLU), enhancing the model's ability to capture complex patterns in the data.

[4]:
```
# GraphLayer: Processes embeddings via graph structure using adjacency matrix.
class GraphLayer(nn.Module):
    def __init__(self, adj, code_size, graph_size):
        super().__init__()
        self.adj = torch.tensor(adj, dtype=torch.float32)  # Convert numpy␣
↪array to tensor
        self.dense = nn.Linear(code_size, graph_size)  # Dense layer for␣
↪transforming embeddings
        self.activation = nn.LeakyReLU()  # Activation function to introduce␣
↪non-linearity

    def forward(self, code_x, neighbor, c_embeddings, n_embeddings):
        # Apply unsqueeze to expand dimensions for matrix operations
```

```
        center_codes = torch.unsqueeze(code_x, dim=-1)
        neighbor_codes = torch.unsqueeze(neighbor, dim=-1)

        # Compute embeddings based on input codes and their neighbors
        center_embeddings = center_codes * c_embeddings
        neighbor_embeddings = neighbor_codes * n_embeddings
        # Multiply embeddings by adjacency matrix to propagate through the graph
        cc_embeddings = center_codes * torch.matmul(self.adj, center_embeddings)
        cn_embeddings = center_codes * torch.matmul(self.adj,␣
↪neighbor_embeddings)
        nn_embeddings = neighbor_codes * torch.matmul(self.adj,␣
↪neighbor_embeddings)
        nc_embeddings = neighbor_codes * torch.matmul(self.adj,␣
↪center_embeddings)

        # Combine embeddings and pass through dense layer with activation
        co_embeddings = self.activation(self.dense(center_embeddings +␣
↪cc_embeddings + cn_embeddings))
        no_embeddings = self.activation(self.dense(neighbor_embeddings +␣
↪nn_embeddings + nc_embeddings))
        return co_embeddings, no_embeddings
```

TransitionLayer

- Purpose: The TransitionLayer manages dynamic changes in the graph structure, reflecting transitions over time or between states. This layer is essential for temporal or dynamic graph models where the state of the graph changes in a way that is significant for the task (e.g., temporal changes in a patient's health records).
- Functionality:
  - State Update with GRU: It uses a GRUCell for updating the hidden states based on the current embeddings. GRUs are effective in managing sequences where the current output is dependent on previous states, making them suitable for time-series data or any data with temporal dynamics.
  - Attention Mechanism: Incorporates a single-head attention mechanism to selectively focus on important features from the embeddings. This is particularly useful in complex scenarios where not all parts of the input are equally relevant to the output.
  - Dynamic Handling of Divisions: It processes divided inputs (perhaps representing different categories or types of inputs) and manages transitions based on these divisions, reflecting complex internal dynamics within the data.

[5]:
```
# TransitionLayer: Manages state transitions in graph sequences using GRU and␣
↪attention.
class TransitionLayer(nn.Module):
    def __init__(self, code_num, graph_size, hidden_size, t_attention_size,␣
↪t_output_size):
        super().__init__()
```

```
        self.gru = nn.GRUCell(input_size=graph_size, hidden_size=hidden_size) ␣
↪# GRU cell for state transitions
        # Attention layer for processing graph-level information
        self.single_head_attention = SingleHeadAttentionLayer(graph_size,␣
↪graph_size, t_output_size, t_attention_size)
        self.activation = nn.Tanh()  # Tanh activation for smooth non-linearity

        self.code_num = code_num  # Total number of codes
        self.hidden_size = hidden_size  # Dimension of hidden state

    def forward(self, t, co_embeddings, divided, no_embeddings,␣
↪unrelated_embeddings, hidden_state=None):
        # Process middle states based on input divisions
        m1, m2, m3 = divided[:, 0], divided[:, 1], divided[:, 2]
        # Find indices where division values are positive
        m1_index = torch.where(m1 > 0)[0]
        m2_index = torch.where(m2 > 0)[0]
        m3_index = torch.where(m3 > 0)[0]
        # Initialize new hidden state for all codes
        h_new = torch.zeros((self.code_num, self.hidden_size),␣
↪dtype=co_embeddings.dtype).to(co_embeddings.device)
        output_m1 = 0
        output_m23 = 0
        # Compute new state for m1 divisions
        if len(m1_index) > 0:
            m1_embedding = co_embeddings[m1_index]
            h = hidden_state[m1_index] if hidden_state is not None else None
            h_m1 = self.gru(m1_embedding, h)
            h_new[m1_index] = h_m1
            output_m1, _ = torch.max(h_m1, dim=-2)
        # Compute new state for m2 and m3 divisions if t > 0
        if t > 0 and len(m2_index) + len(m3_index) > 0:
            # Combine embeddings for m2 and m3 indices for attention processing
            q = torch.vstack([no_embeddings[m2_index],␣
↪unrelated_embeddings[m3_index]])
            v = torch.vstack([co_embeddings[m2_index], co_embeddings[m3_index]])

            # Process combined embeddings through the attention layer
            h_m23 = self.activation(self.single_head_attention(q, q, v))

            # Update the hidden states for m2 and m3 indices based on attention␣
↪outputs
            h_new[m2_index] = h_m23[:len(m2_index)]
            h_new[m3_index] = h_m23[len(m2_index):]
```

```
            # Determine the maximum output across m2 and m3 for use in the␣
↪final output
            output_m23, _ = torch.max(h_m23, dim=-2)

        # Determine the final output based on available indices
        if len(m1_index) == 0:
            output = output_m23
        elif len(m2_index) + len(m3_index) == 0:
            output = output_m1
        else:
            # If both m1 and m2/m3 indices have outputs, take the maximum␣
↪across both
            output, _ = torch.max(torch.vstack([output_m1, output_m23]), dim=-2)

        # Return the final aggregated output and the updated hidden states
        return output, h_new
```

Component Description	
Classifier	- **Purpose**: Serves as the final classification layer in the model pipeline.- **Components**: Consists of a linear layer, optional activation function, and a dropout layer.- **Functionality**: Takes the output from the preceding layers, applies dropout for regularization, transforms it through a linear operation, and finally applies an activation function if provided. This setup is typical for neural network classifiers to make final predictions.

[6]:
```
# Classifier: A simple neural network module for classification tasks.
class Classifier(nn.Module):
    def __init__(self, input_size, output_size, dropout_rate=0.,␣
↪activation=None):
        super().__init__()
        self.linear = nn.Linear(input_size, output_size)  # Linear␣
↪transformation to the output size
        self.activation = activation  # Optional activation function
        self.dropout = nn.Dropout(p=dropout_rate)  # Dropout layer to prevent␣
↪overfitting

    def forward(self, x):
        output = self.dropout(x)  # Apply dropout to the input
        output = self.linear(output)  # Apply linear transformation
        if self.activation is not None:
            output = self.activation(output)  # Apply activation function if␣
↪provided
        return output  # Return the final output
```

Component	Description
Chet	- **Purpose**: Integrates various specialized layers to process graph-structured data and sequences.- **Components**: Includes `EmbeddingLayer`, `GraphLayer`, `TransitionLayer`, `DotProductAttention`, and the `Classifier`.- **Functionality**: Orchestrates the flow of data through multiple layers designed to handle embeddings, graph interactions, transitions in dynamic states, and finally classification.

| **forward (Chet)** | - **Purpose**: Defines how data passes through the model during the forward pass.- **Functionality**:

Obtains embeddings from `EmbeddingLayer`.

Processes these through `GraphLayer` for each sequence element.

Uses `TransitionLayer` to manage transitions and update states based on dynamic graph elements.

Applies `DotProductAttention` to aggregate sequence data effectively.

Feeds the aggregated output into `Classifier` for final predictions.

[7]:
```
# Model: Main model integrating various components for processing graphs.
class Chet(nn.Module):
    def __init__(self, code_num, code_size, adj, graph_size, hidden_size,
                 t_attention_size, t_output_size, output_size, dropout_rate,␣
 ↪activation):
        super().__init__()
        # Initialize embedding, graph, and transition layers
        self.embedding_layer = EmbeddingLayer(code_num, code_size, graph_size)
        self.graph_layer = GraphLayer(adj, code_size, graph_size)
        self.transition_layer = TransitionLayer(code_num, graph_size,␣
 ↪hidden_size, t_attention_size, t_output_size)
        self.attention = DotProductAttention(hidden_size, 32)  # Dot product␣
 ↪attention mechanism
        self.classifier = Classifier(hidden_size, output_size, dropout_rate,␣
 ↪activation)  # Classifier component

    def forward(self, code_x, divided, neighbors, lens):
        # Generate embeddings from the embedding layer
        embeddings = self.embedding_layer()
        c_embeddings, n_embeddings, u_embeddings = embeddings
        output = []
        # Process each sequence in the batch
        for code_x_i, divided_i, neighbor_i, len_i in zip(code_x, divided,␣
 ↪neighbors, lens):
            no_embeddings_i_prev = None  # Store previous neighbor embeddings
            output_i = []
            h_t = None  # Initialize hidden state
```

```
            # Iterate over time steps within a sequence
            for t, (c_it, d_it, n_it, len_it) in enumerate(zip(code_x_i,␣
↪divided_i, neighbor_i, range(len_i))):
                # Process current time step using the graph layer
                co_embeddings, no_embeddings = self.graph_layer(c_it, n_it,␣
↪c_embeddings, n_embeddings)
                # Transition layer updates based on current and previous states
                output_it, h_t = self.transition_layer(t, co_embeddings, d_it,␣
↪no_embeddings_i_prev, u_embeddings, h_t)
                no_embeddings_i_prev = no_embeddings  # Update previous␣
↪embeddings
                output_i.append(output_it)  # Collect outputs for each time step
            # Apply attention to the sequence of outputs
            output_i = self.attention(torch.vstack(output_i))
            output.append(output_i)  # Collect final outputs for all sequences

        output = torch.vstack(output)  # Stack all sequence outputs
        output = self.classifier(output)  # Classify the aggregated outputs
        return output  # Return the final model output
```

Component	Sub-Component	Purpose	Functionality
Chet_Transformer	**Embedding Layer**	To initialize code embeddings	Converts `code_num`, `code_size`, and `graph_size` into dense vector representations of the input codes, facilitating easier and more effective processing by neural networks.
	Graph Layer	To process graph-based data	Uses the adjacency matrix (`adj`) along with embeddings to process relationships between different codes in a graph-structured data setup.
	Linear Transition Layer	To transform graph layer outputs	Maps the output from the `GraphLayer` (of size `graph_size`) to a higher dimensional space (`hidden_size`) using a linear transformation, followed by a `tanh` activation to introduce non-linearity. **This layer is to replace the original Transition layer.**
	Dot Product Attention	To emphasize important features	Applies a dot product attention mechanism on the sequence of outputs, focusing on significant features that are crucial for the prediction task.
	Classifier	To classify the aggregated outputs	Uses the output from the attention mechanism to make final classifications. Includes dropout for regularization and uses the specified activation function for the final output layer.

Component	Sub-Component	Purpose	Functionality
Forward	**Data Processing Loop**	To handle sequence processing per batch	Iterates over each sequence in the batch, processing each time step's data through the graph layer, followed by the transition layer, and collects outputs for attention processing.
	Attention and Classification	To aggregate and classify outputs	Aggregates the transformed outputs using attention, and then classifies these aggregated outputs using the classifier. The final output is stacked and returned as the model's prediction for the batch.

[8]:
```
import torch
from torch import nn

class Chet_TF(nn.Module):
    def __init__(self, code_num, code_size, adj, graph_size, hidden_size,
                 output_size, dropout_rate, activation, t_attention_size,␣
 ↪t_output_size):
        super().__init__()
        # Initialize embedding, graph, and the linear transition layers
        self.embedding_layer = EmbeddingLayer(code_num, code_size, graph_size)
        self.graph_layer = GraphLayer(adj, code_size, graph_size)
        self.linear_transition = nn.Linear(graph_size, hidden_size)  # Linear␣
 ↪layer to replace transition function
        self.tanh = nn.Tanh()  # Tanh activation function
        self.attention = DotProductAttention(hidden_size, 32)  # Dot product␣
 ↪attention mechanism
        self.classifier = Classifier(hidden_size, output_size, dropout_rate,␣
 ↪activation)  # Classifier component

    def forward(self, code_x, divided, neighbors, lens):
        # Generate embeddings from the embedding layer
        embeddings = self.embedding_layer()
        c_embeddings, n_embeddings, u_embeddings = embeddings
        output = []
        # Process each sequence in the batch
        for code_x_i, divided_i, neighbor_i, len_i in zip(code_x, divided,␣
 ↪neighbors, lens):
            output_i = []
            # Iterate over time steps within a sequence
            for c_it, n_it in zip(code_x_i, neighbor_i):
                # Process current time step using the graph layer
                co_embeddings, no_embeddings = self.graph_layer(c_it, n_it,␣
 ↪c_embeddings, n_embeddings)
                # Apply the linear layer and then tanh activation
```

```
                transformed_output = self.tanh(self.
↪linear_transition(co_embeddings))
                output_i.append(transformed_output)  # Collect outputs for each␣
↪time step
            # Apply attention to the sequence of outputs
            output_i = self.attention(torch.vstack(output_i))
            output.append(output_i)  # Collect final outputs for all sequences
        output = torch.vstack(output)  # Stack all sequence outputs
        output = self.classifier(output)  # Classify the aggregated outputs
        return output  # Return the final model output
```

3.2.2 Training Step

- Historical Data Feature: Implements a feature extraction method historical_hot to transform code data based on historical visits, which could be a critical feature depending on the task.
- Dynamic Model Handling: Allows flexibility in model choice and parameters, facilitating easy experiments with different configurations.
- Robust Training and Validation Mechanics: Incorporates modern training enhancements like dynamic learning rate adjustments and early stopping based on validation performance to optimize training outcomes.
- Resource Management: Carefully manages device resources, ensuring that all operations are performed on the designated computing device (GPU/CPU).

[9]:
```
import os
import random
import time

import torch
import numpy as np

from utils import load_adj, EHRDataset, format_time, MultiStepLRScheduler
from metrics import evaluate_codes, evaluate_hf


def historical_hot(code_x, code_num, lens):
    result = np.zeros((len(code_x), code_num), dtype=int)
    for i, (x, l) in enumerate(zip(code_x, lens)):
        result[i] = x[l - 1]
    return result


def train(model_name, dropout_rate, epochs, code_size, graph_size, hidden_size,␣
↪t_attention_size, t_output_size, batch_size):
    seed = 1000
    dataset = 'mimic3'  # 'mimic3' or 'eicu'
    task = 'h'  # 'm' or 'h'
    model_name = model_name
```

```
    use_cuda = True
    device = torch.device('cuda' if torch.cuda.is_available() and use_cuda else␣
↪'cpu')

    code_size = code_size
    graph_size = graph_size
    hidden_size = hidden_size  # rnn hidden size
    t_attention_size = t_attention_size
    t_output_size = hidden_size
    batch_size = batch_size
    epochs = epochs

    code_size = 48
    graph_size = 32
    hidden_size = 150  # rnn hidden size
    t_attention_size = 32
    t_output_size = hidden_size
    batch_size = 32
    epochs = 5

    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)

    dataset_path = os.path.join('data', dataset, 'standard')
    train_path = os.path.join(dataset_path, 'train')
    valid_path = os.path.join(dataset_path, 'valid')
    test_path = os.path.join(dataset_path, 'test')

    code_adj = load_adj(dataset_path, device=device)
    code_num = len(code_adj)
    print('loading train data ...')
    train_data = EHRDataset(train_path, label=task, batch_size=batch_size,␣
↪shuffle=True, device=device)
    print('loading valid data ...')
    valid_data = EHRDataset(valid_path, label=task, batch_size=batch_size,␣
↪shuffle=False, device=device)
    print('loading test data ...')
    test_data = EHRDataset(test_path, label=task, batch_size=batch_size,␣
↪shuffle=False, device=device)

    test_historical = historical_hot(valid_data.code_x, code_num, valid_data.
↪visit_lens)

    task_conf = {
        'h': {
```

```
            'dropout': 0.0,
            'output_size': 1,
            'evaluate_fn': evaluate_hf,
            'lr': {
                'init_lr': 0.01,
                'milestones': [2, 3, 4],
                'lrs': [1e-3, 1e-4, 1e-5]
            }
        }
    }

    model_select = {
        'Chet': Chet,
        'Chet_TF': Chet_TF,
    }

    output_size = task_conf[task]['output_size']
    activation = torch.nn.Sigmoid()
    loss_fn = torch.nn.BCELoss()
    evaluate_fn = task_conf[task]['evaluate_fn']
    dropout_rate = task_conf[task]['dropout']

    param_path = os.path.join('data', 'params', dataset, task, model_name)
    if not os.path.exists(param_path):
        os.makedirs(param_path)

    # model = Chet(code_num=code_num, code_size=code_size,
    #              adj=code_adj, graph_size=graph_size,␣
↪hidden_size=hidden_size, t_attention_size=t_attention_size,
    #              t_output_size=t_output_size,
    #              output_size=output_size, dropout_rate=dropout_rate,␣
↪activation=activation).to(device)

    model = model_select[model_name](code_num=code_num, code_size=code_size,
                     adj=code_adj, graph_size=graph_size,␣
↪hidden_size=hidden_size, t_attention_size=t_attention_size,
                     t_output_size=t_output_size,
                     output_size=output_size, dropout_rate=dropout_rate,␣
↪activation=activation).to(device)

    optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
    scheduler = MultiStepLRScheduler(optimizer, epochs,␣
↪task_conf[task]['lr']['init_lr'],
                                     task_conf[task]['lr']['milestones'],␣
↪task_conf[task]['lr']['lrs'])
```

```
    pytorch_total_params = sum(p.numel() for p in model.parameters() if p.
↪requires_grad)
    print(pytorch_total_params)

    best_val_loss = float('inf')  # Initialize to a large number
    best_epoch = 0  # Track the epoch at which the best model was found
    for epoch in range(epochs):
        print('Epoch %d / %d:' % (epoch + 1, epochs))
        model.train()
        total_loss = 0.0
        total_num = 0
        steps = len(train_data)
        st = time.time()
        scheduler.step()
        for step in range(len(train_data)):
            optimizer.zero_grad()
            code_x, visit_lens, divided, y, neighbors = train_data[step]
            output = model(code_x, divided, neighbors, visit_lens).squeeze()
            loss = loss_fn(output, y)
            loss.backward()
            optimizer.step()
            total_loss += loss.item() * output_size * len(code_x)
            total_num += len(code_x)

            end_time = time.time()
            remaining_time = format_time((end_time - st) / (step + 1) * (steps␣
↪- step - 1))
            print('\r    Step %d / %d, remaining time: %s, loss: %.4f'
                  % (step + 1, steps, remaining_time, total_loss /␣
↪total_num), end='')
        train_data.on_epoch_end()
        et = time.time()
        time_cost = format_time(et - st)
        print('\r    Step %d / %d, time cost: %s, loss: %.4f' % (steps, steps,␣
↪time_cost, total_loss / total_num))

        # Evaluate the model on validation data
        valid_loss, f1_score = evaluate_fn(model, valid_data, loss_fn,␣
↪output_size, test_historical)
        # Save the model only if the validation loss improved
        if valid_loss < best_val_loss:
            best_val_loss = valid_loss
            best_epoch = epoch
            model_save_path = os.path.join(param_path, 'best_model.pt')
            torch.save(model.state_dict(), model_save_path)
            print(f'Best model saved at epoch {epoch + 1} with validation loss:␣
↪{valid_loss:.4f}')
```

```
    # print out the best epoch and its performance after training is complete
    print(f'Best performing model was at epoch {best_epoch + 1} with validation␣
 ↪loss: {best_val_loss:.4f}')

    return model
```

- Load the Full Chet model if it's available and store.
- Train the Full Chet model if it's not available.

[10]:
```
from models.model import Model

model_name = 'Chet'
model_path = os.path.join('data', 'params', 'mimic3', 'h', model_name,␣
 ↪'best_model.pt')
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
code_size = 48
graph_size = 32
hidden_size = 150  # rnn hidden size
t_attention_size = 32
t_output_size = hidden_size
batch_size = 32
epochs = 5
output_size = 1
activation = torch.nn.Sigmoid()
dropout_rate = 0.0

#model_Chet = train(model_name, dropout_rate, epochs, code_size, graph_size,␣
 ↪hidden_size, t_attention_size, t_output_size, batch_size)
model_Chet = Model(code_num=code_num, code_size=code_size,
                   adj=code_adj, graph_size=graph_size, hidden_size=hidden_size,␣
 ↪t_attention_size=t_attention_size,
                   t_output_size=t_output_size,
                   output_size=output_size, dropout_rate=dropout_rate,␣
 ↪activation=activation).to(device)

if os.path.exists(model_path):
    # Load the model if it exists
    model_Chet.load_state_dict(torch.load(model_path))
    model_Chet.to(device)
    print(f'Model loaded from {model_path}')
else:
    # Train the model if the file does not exist
    # Assuming train_data, optimizer, and loss_fn are defined elsewhere
    print('Model is not available, starting training...')
```

```
    model_Chet = train(model_name, dropout_rate, epochs, code_size, graph_size,␣
 ↪hidden_size, t_attention_size, t_output_size, batch_size)

print(model_Chet)
total_params = sum(p.numel() for p in model_Chet.parameters() if p.
 ↪requires_grad)
print(f"Total trainable parameters: {total_params}")
for name, param in model_Chet.named_parameters():
    if param.requires_grad:
        print(f"Layer: {name} | Size: {param.size()} | Total parameters: {param.
 ↪numel()}")
```

```
Model loaded from data/params/mimic3/h/Chet/best_model.pt
Model(
  (embedding_layer): EmbeddingLayer()
  (graph_layer): GraphLayer(
    (dense): Linear(in_features=48, out_features=32, bias=True)
    (activation): LeakyReLU(negative_slope=0.01)
  )
  (transition_layer): TransitionLayer(
    (gru): GRUCell(32, 150)
    (single_head_attention): SingleHeadAttentionLayer(
      (dense_q): Linear(in_features=32, out_features=32, bias=True)
      (dense_k): Linear(in_features=32, out_features=32, bias=True)
      (dense_v): Linear(in_features=32, out_features=150, bias=True)
    )
    (activation): Tanh()
  )
  (attention): DotProductAttention(
    (dense): Linear(in_features=150, out_features=32, bias=True)
  )
  (classifier): Classifier(
    (linear): Linear(in_features=150, out_features=1, bias=True)
    (activation): Sigmoid()
    (dropout): Dropout(p=0.0, inplace=False)
  )
)
Total trainable parameters: 170813
Layer: embedding_layer.c_embeddings | Size: torch.Size([581, 48]) | Total
parameters: 27888
Layer: embedding_layer.n_embeddings | Size: torch.Size([581, 48]) | Total
parameters: 27888
Layer: embedding_layer.u_embeddings | Size: torch.Size([581, 32]) | Total
parameters: 18592
Layer: graph_layer.dense.weight | Size: torch.Size([32, 48]) | Total parameters:
1536
Layer: graph_layer.dense.bias | Size: torch.Size([32]) | Total parameters: 32
```

```
Layer: transition_layer.gru.weight_ih | Size: torch.Size([450, 32]) | Total
parameters: 14400
Layer: transition_layer.gru.weight_hh | Size: torch.Size([450, 150]) | Total
parameters: 67500
Layer: transition_layer.gru.bias_ih | Size: torch.Size([450]) | Total
parameters: 450
Layer: transition_layer.gru.bias_hh | Size: torch.Size([450]) | Total
parameters: 450
Layer: transition_layer.single_head_attention.dense_q.weight | Size:
torch.Size([32, 32]) | Total parameters: 1024
Layer: transition_layer.single_head_attention.dense_q.bias | Size:
torch.Size([32]) | Total parameters: 32
Layer: transition_layer.single_head_attention.dense_k.weight | Size:
torch.Size([32, 32]) | Total parameters: 1024
Layer: transition_layer.single_head_attention.dense_k.bias | Size:
torch.Size([32]) | Total parameters: 32
Layer: transition_layer.single_head_attention.dense_v.weight | Size:
torch.Size([150, 32]) | Total parameters: 4800
Layer: transition_layer.single_head_attention.dense_v.bias | Size:
torch.Size([150]) | Total parameters: 150
Layer: attention.context | Size: torch.Size([32, 1]) | Total parameters: 32
Layer: attention.dense.weight | Size: torch.Size([32, 150]) | Total parameters:
4800
Layer: attention.dense.bias | Size: torch.Size([32]) | Total parameters: 32
Layer: classifier.linear.weight | Size: torch.Size([1, 150]) | Total parameters:
150
Layer: classifier.linear.bias | Size: torch.Size([1]) | Total parameters: 1
```

- Load the Chet_TF model if it's available and store.
- Train the Chet_TF model if it's not available.

[11]:
```
model_name = 'Chet_TF'
model_path = os.path.join('data', 'params', 'mimic3', 'h', model_name,␣
 ↪'best_model.pt')
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
code_size = 48
graph_size = 32
hidden_size = 16  # rnn hidden size
t_attention_size = 32
t_output_size = hidden_size
batch_size = 32
epochs = 5
output_size = 1
activation = torch.nn.Sigmoid()
dropout_rate = 0.0

#model_tf = train(model_name, dropout_rate, epochs, code_size, graph_size,␣
 ↪hidden_size, t_attention_size, t_output_size, batch_size)
```

```
model_tf = Chet_TF(code_num=code_num, code_size=code_size,
                   adj=code_adj, graph_size=graph_size,␣
  ↪hidden_size=hidden_size, t_attention_size=t_attention_size,
                   t_output_size=t_output_size,
                   output_size=output_size, dropout_rate=dropout_rate,␣
  ↪activation=activation).to(device)


if os.path.exists(model_path):
    # Load the model if it exists
    model_tf.load_state_dict(torch.load(model_path))
    model_tf.to(device)
    print(f'Model loaded from {model_path}')
else:
    # Train the model if the file does not exist
    # Assuming train_data, optimizer, and loss_fn are defined elsewhere
    print('Model is not available, starting training...')
    model_tf = train(model_name, dropout_rate, epochs, code_size, graph_size,␣
  ↪hidden_size, t_attention_size, t_output_size, batch_size)

print(model_tf)
total_params = sum(p.numel() for p in model_tf.parameters() if p.requires_grad)
print(f"Total trainable parameters: {total_params}")
for name, param in model_tf.named_parameters():
    if param.requires_grad:
        print(f"Layer: {name} | Size: {param.size()} | Total parameters: {param.
  ↪numel()}")
```

```
Model loaded from data/params/mimic3/h/Chet_TF/best_model.pt
Chet_TF(
  (embedding_layer): EmbeddingLayer()
  (graph_layer): GraphLayer(
    (dense): Linear(in_features=48, out_features=32, bias=True)
    (activation): LeakyReLU(negative_slope=0.01)
  )
  (linear_transition): Linear(in_features=32, out_features=16, bias=True)
  (tanh): Tanh()
  (attention): DotProductAttention(
    (dense): Linear(in_features=16, out_features=32, bias=True)
  )
  (classifier): Classifier(
    (linear): Linear(in_features=16, out_features=1, bias=True)
    (activation): Sigmoid()
    (dropout): Dropout(p=0.0, inplace=False)
  )
)
```

```
Total trainable parameters: 77057
Layer: embedding_layer.c_embeddings | Size: torch.Size([581, 48]) | Total
parameters: 27888
Layer: embedding_layer.n_embeddings | Size: torch.Size([581, 48]) | Total
parameters: 27888
Layer: embedding_layer.u_embeddings | Size: torch.Size([581, 32]) | Total
parameters: 18592
Layer: graph_layer.dense.weight | Size: torch.Size([32, 48]) | Total parameters:
1536
Layer: graph_layer.dense.bias | Size: torch.Size([32]) | Total parameters: 32
Layer: linear_transition.weight | Size: torch.Size([16, 32]) | Total parameters:
512
Layer: linear_transition.bias | Size: torch.Size([16]) | Total parameters: 16
Layer: attention.context | Size: torch.Size([32, 1]) | Total parameters: 32
Layer: attention.dense.weight | Size: torch.Size([32, 16]) | Total parameters:
512
Layer: attention.dense.bias | Size: torch.Size([32]) | Total parameters: 32
Layer: classifier.linear.weight | Size: torch.Size([1, 16]) | Total parameters:
16
Layer: classifier.linear.bias | Size: torch.Size([1]) | Total parameters: 1
```

3.3 Results

Evaluate the two models Chet Full Model (Chet) and Chet Without TF (Chet_TF).

[12]:
```python
from metrics import evaluate_hf

dataset_path = os.path.join('data', dataset, 'standard')
test_path = os.path.join(dataset_path, 'test')
test_data = EHRDataset(test_path, label='h', batch_size=32, shuffle=False,␣
 ↪device=device)
loss_fn = torch.nn.BCELoss()
output_size = 1
code_adj = load_adj(dataset_path, device=device)
code_num = len(code_adj)
test_historical = historical_hot(test_data.code_x, code_num, test_data.
 ↪visit_lens)


test_loss_chet_tf, f1_score_chet_tf = evaluate_hf(model_Chet, test_data,␣
 ↪loss_fn, output_size, test_historical)
test_loss_tf, f1_score_tf = evaluate_hf(model_tf, test_data, loss_fn,␣
 ↪output_size, test_historical)

# Print the evaluation results
print(f'Chet model test loss: {test_loss_chet_tf:.4f}, F1 score:␣
 ↪{f1_score_chet_tf:.4f}')
```

```
print(f'Chet_TF model test loss: {test_loss_tf:.4f}, F1 score: {f1_score_tf:.
 ↪4f}')
```

```
    Evaluating step 1 / 1
outputs:  [array([0.3294184 , 0.33162332, 0.3214092 , 0.32166973, 0.31956545,
       0.32273805, 0.3161934 , 0.31994167, 0.33121592, 0.32081184,
       0.33101666, 0.32222876, 0.31928122, 0.3313369 , 0.32103992,
       0.3322964 , 0.33153108, 0.3212745 , 0.3200318 , 0.32216108],
      dtype=float32)]
    Evaluation: loss: 0.6381 --- auc: 1.0000 --- f1_score: 0.0000
    Evaluating step 1 / 1
outputs:  [array([0.4656841 , 0.46569526, 0.46569616, 0.46569756, 0.46569523,
       0.46567976, 0.46566528, 0.4656959 , 0.46568787, 0.46568245,
       0.4656814 , 0.46569097, 0.46569672, 0.4656968 , 0.46570364,
       0.46569195, 0.46568668, 0.46569166, 0.4656942 , 0.4656991 ],
      dtype=float32)]
    Evaluation: loss: 0.6749 --- auc: 0.3626 --- f1_score: 0.0000
Chet model test loss: 0.6381, F1 score: 0.0000
Chet_TF model test loss: 0.6749, F1 score: 0.0000
```

3.3.1 Model comparison

Based on the metrics obtained from testing, the Full Chet model demonstrated superior performance, exhibiting a lower loss value compared to the variant without the transition layer. This suggests that the inclusion of the transition layer in the Full Chet model may contribute significantly to its ability to more effectively minimize errors in predictions.

3.4 Discussion

- **Based on the outcomes observed thus far:**

1. The Full Chet model outperforms the variant lacking the transition layer, corroborating the findings reported in the original paper [1].
2. The reproducibility of the results from the original paper [1] is affirmed due to the accessibility of both the data and the model.
3. Constructing three levels of graph embeddings from global diagnostic codes presented significant challenges. Nevertheless, the availability of the original author's code facilitated understanding and reproduction of these methods.

- **Areas for Improvement:**

1. Currently, the dataset in use comprises a demo set with only 10 patients. To augment the dataset, this data was replicated ten times, resulting in 100 patient records.
2. The intention is to employ the actual MIMIC-III dataset for retraining the model once it becomes accessible.
3. The control model, a CNN-based approach known as Deepr [2], has not yet been implemented. Its integration is planned for subsequent phases to enable a more comprehensive analysis of results.
4. There is a plan to refactor the code to leverage more widely-used packages such as DataLoader, enhancing its efficiency and readability.

5. Efforts are underway to refine the code further, contributing to the pyHealth project, which aims to provide robust health informatics solutions.

4 References

[1] Lu, Chang, Tian Han, and Yue Ning. "Context-aware health event prediction via transition functions on dynamic disease graphs." In Proceedings of the AAAI Conference on Artificial Intelligence, vol. 36, no. 4, pp. 4567-4574. 2022. [2] Phuoc Nguyen, Truyen Tran, Nilmini Wickramasinghe and Svetha Venkatesh, " Deepr : A Convolutional Net for Medical Records," in IEEE Journal of Biomedical and Health Informatics, vol. 21, no. 1, pp. 22-30, Jan. 2017.