Project Draft

Monisha Vatikuti

mav12@illinois.edu

Team 126

Paper 16 - Context-aware Health Event Prediction via Transition Functions on Dynamic Disease Graphs

Github Repo: https://github.com/mvattiku/cs598-dlh-project

Introduction

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

- Background of the problem
  - what type of problem: disease/readmission/mortality prediction, feature engineeing, data processing, etc
  - what is the importance/meaning of solving the problem
  - what is the difficulty of the problem
  - the state of the art methods and effectiveness.
- Paper explanation
  - what did the paper propose
  - what is the innovations of the method
  - how well the proposed method work (in its own metrics)
  - what is the contribution to the reasearch regime (referring the Background above, how important the paper is to the problem).

Here we are trying to reproduce the new model called Chet that is proposed in the Context-aware Health Event Prediction via Transition Functions on Dynamic Disease Graphs paper. The paper is trying to address the issue of viewing histocial diagnosis from EHR data as independent events which disregards potential relationships of diseases among visits. Most deep learning models for disease classification using longitudinal EHR data treat disease diagnoses as independent events within individual visits. But there exist patterns in the co-occurrence of disease diagnoses that could be valuable for predicting future patient outcomes. And these patterns are being ignored in current set of models. So the paper proposes a novel deep learning model called Chet (context-aware health event prediction via transition functions on dynamic disease graphs) which looks at both the evolution of diseases and the relationships between diseases to predict future diagnoses. Chet aims to look for interlinked patterns by trying to learn how diseases are progressing over consecutive visits to anticipate future diagnoses.

In particular, Chet model learns the evolution of diagnosed diseases across a patient's doctor visits and exploits this learned disease context to forecast future outcomes and diagnoses. The key innovation of this approach lies in its incorporation of both disease co-occurrence information and the dynamic nature of diseases into the model. To accomplish this, the model constructs a weighted disease combination based on the entire longitudinal EHR data globally, as well as a disease subgraph based on the specific visit locally. To account for the dynamic nature of diseases, the paper employs disease-level temporal learning with multiple diagnosis roles and corresponding transition functions to capture historical contexts.

Scope of Reproducibility:

In this replication study, I will adopt the same methodology proposed in the paper for data selection, cleaning, and preprocessing. I will be using the MIMIC-III and MIMIC-IV datasets and randomly divide the data into training, validation, and test sets as done in the original study. Then I will build the diagnosis graphs and compute the adjacency matrices for their corresponding subgraphs using the same steps as outlined in the paper. Then I will train the model for both diagnosis prediction and heart failure prediction and compare the metrics.

Methodology

This methodology is the core of your project. It consists of run-able codes with necessary annotations to show the expeiment you executed for testing the hypotheses.

The methodology at least contains two subsections **data** and **model** in your experiment.

Data

Data includes raw data (MIMIC III tables), descriptive statistics (our homework questions), and data processing (feature engineering).

- Source of the data: Currently I am using **MIMIC-III Clinical Database Demo**. I download this data from https://physionet.org/content/mimiciii-demo/1.4/ and is located in this repo in the following directory `data/mimic3/raw`. But for the final project I will pull the full dataset.
- Data process: I am processing the data the same way as proposed in the paper. I have pulled the original scripts from papers repo (located under `Chet/` directory). And I am running the `run_preprocess.py` script on the dataset.
- Statistics: The output of the followin script prints out the necessary statistics.

In [1]:
```python
# This function is copied from the original code (run_prepocess.py script)
# This function also uses some scripts from preprocess module which has also been copied from the source code
import os
from sys import exit
import _pickle as pickle

from preprocess import save_sparse, save_data
from preprocess.parse_csv import Mimic3Parser, Mimic4Parser, EICUParser
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

def run_preprocess_modified(dataset='mimic3', from_saved=True):
    conf = {
        'mimic3': {
            'parser': Mimic3Parser,
            'train_num': 6000,
            'test_num': 1000,
            'threshold': 0.01
        },
        'mimic4': {
            'parser': Mimic4Parser,
            'train_num': 8000,
            'test_num': 1000,
            'threshold': 0.01,
            'sample_num': 10000
        },
        'eicu': {
            'parser': EICUParser,
            'train_num': 8000,
            'test_num': 1000,
            'threshold': 0.01
        }
    }
    # from_saved = True
    data_path = 'data'
    # dataset = 'mimic3'  # mimic3, eicu, or mimic4
    dataset_path = os.path.join(data_path, dataset)
    raw_path = os.path.join(dataset_path, 'raw')
    # if not os.path.exists(raw_path):
    #     os.makedirs(raw_path)
    #     print('please put the CSV files in `data/%s/raw`' % dataset)
    #     exit()
    parsed_path = os.path.join(dataset_path, 'parsed')
    if from_saved:
        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:
        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'))

    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)

    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)

    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'))

    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'])

    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)

    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)
```

In [2]:
```python
run_preprocess_modified(dataset='mimic3', from_saved=True)
```

```
patient num: 7493
max admission num: 42
mean admission num: 2.66
max code num in an admission: 39
mean code num in an admission: 13.06
encoding code ...
There are 4880 codes
generating code levels ...
	100%00%
There are 6000 train, 493 valid, 1000 test samples
generating code code adjacent matrix ...
	6000 / 6000
building train codes features and labels ...
	6000 / 6000
building valid codes features and labels ...
	493 / 493
building test codes features and labels ...
	1000 / 1000
generating train neighbors ...
	6000 / 6000
generating valid neighbors ...
	493 / 493
generating test neighbors ...
	1000 / 1000
generating train middles ...
	6000 / 6000
generating valid middles ...
	493 / 493
generating test middles ...
	1000 / 1000
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
```

Model

The model classes and functions have been pulled from the papers original code

In [3]:
```python
import torch
import numpy as np
import random
from torch import nn
import math
```

1. Graph Layer

This is the Optimized dynamic graph layer which extracts both local and global contexts for diagnoses and neighbors in visit t and then calculate hidden embeddings for diagnoses and neighbors.

In [4]:
```python
class GraphLayer(nn.Module):
    def __init__(self, adj, code_size, graph_size):
        super().__init__()
        self.adj = adj
        self.dense = nn.Linear(code_size, graph_size)
        self.activation = nn.LeakyReLU()

    def forward(self, code_x, neighbor, c_embeddings, n_embeddings):
        center_codes = torch.unsqueeze(code_x, dim=-1)
        neighbor_codes = torch.unsqueeze(neighbor, dim=-1)

        center_embeddings = center_codes * c_embeddings
        neighbor_embeddings = neighbor_codes * n_embeddings
        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)

        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
```

2. Transition Layer

This is the transition functions layer which takes the hidden embeddings from the graph layer as inputs to this layer and applies GRU, M-GRU, or attention functions.

In [5]:
```python
class SingleHeadAttentionLayer(nn.Module):
    def __init__(self, query_size, key_size, value_size, attention_size):
        super().__init__()
        self.attention_size = attention_size
        self.dense_q = nn.Linear(query_size, attention_size)
        self.dense_k = nn.Linear(key_size, attention_size)
        self.dense_v = nn.Linear(query_size, value_size)

    def forward(self, q, k, v):
        query = self.dense_q(q)
        key = self.dense_k(k)
        value = self.dense_v(v)
        g = torch.div(torch.matmul(query, key.T), math.sqrt(self.attention_size))
        score = torch.softmax(g, dim=-1)
        output = torch.sum(torch.unsqueeze(score, dim=-1) * value, dim=-2)
        return output


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)
        self.single_head_attention = SingleHeadAttentionLayer(graph_size, graph_size, t_output_size, t_attention_size)
        self.activation = nn.Tanh()

        self.code_num = code_num
        self.hidden_size = hidden_size

    def forward(self, t, co_embeddings, divided, no_embeddings, unrelated_embeddings, hidden_state=None):
        m1, m2, m3 = divided[:, 0], divided[:, 1], divided[:, 2]
        m1_index = torch.where(m1 > 0)[0]
        m2_index = torch.where(m2 > 0)[0]
        m3_index = torch.where(m3 > 0)[0]
        h_new = torch.zeros((self.code_num, self.hidden_size), dtype=co_embeddings.dtype).to(co_embeddings.device)
        output_m1 = 0
        output_m23 = 0
        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)
        if t > 0 and len(m2_index) + len(m3_index) > 0:
            q = torch.vstack([no_embeddings[m2_index], unrelated_embeddings[m3_index]])
            v = torch.vstack([co_embeddings[m2_index], co_embeddings[m3_index]])
            h_m23 = self.activation(self.single_head_attention(q, q, v))
            h_new[m2_index] = h_m23[:len(m2_index)]
            h_new[m3_index] = h_m23[len(m2_index):]
            output_m23, _ = torch.max(h_m23, dim=-2)
        if len(m1_index) == 0:
            output = output_m23
        elif len(m2_index) + len(m3_index) == 0:
            output = output_m1
        else:
            output, _ = torch.max(torch.vstack([output_m1, output_m23]), dim=-2)
        return output, h_new
```

3. Embedding Layer

This is the embedding layer.

In [6]:
```python
class EmbeddingLayer(nn.Module):
    def __init__(self, code_num, code_size, graph_size):
        super().__init__()
        self.code_num = code_num
        self.c_embeddings = nn.Parameter(data=nn.init.xavier_uniform_(torch.empty(code_num, code_size)))
        self.n_embeddings = nn.Parameter(data=nn.init.xavier_uniform_(torch.empty(code_num, code_size)))
        self.u_embeddings = nn.Parameter(data=nn.init.xavier_uniform_(torch.empty(code_num, graph_size)))

    def forward(self):
        return self.c_embeddings, self.n_embeddings, self.u_embeddings
```

4. Classifier and Model

In [7]:
```python
class DotProductAttention(nn.Module):
    def __init__(self, value_size, attention_size):
        super().__init__()
        self.attention_size = attention_size
        self.context = nn.Parameter(data=nn.init.xavier_uniform_(torch.empty(attention_size, 1)))
        self.dense = nn.Linear(value_size, attention_size)

    def forward(self, x):
        t = self.dense(x)
        vu = torch.matmul(t, self.context).squeeze()
        score = torch.softmax(vu, dim=-1)
        output = torch.sum(x * torch.unsqueeze(score, dim=-1), dim=-2)
        return output


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)
        self.activation = activation
        self.dropout = nn.Dropout(p=dropout_rate)

    def forward(self, x):
        output = self.dropout(x)
        output = self.linear(output)
        if self.activation is not None:
            output = self.activation(output)
        return output


class Model(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__()
        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)
        self.classifier = Classifier(hidden_size, output_size, dropout_rate, activation)

    def forward(self, code_x, divided, neighbors, lens):
        embeddings = self.embedding_layer()
        c_embeddings, n_embeddings, u_embeddings = embeddings
        output = []
        for code_x_i, divided_i, neighbor_i, len_i in zip(code_x, divided, neighbors, lens):
            no_embeddings_i_prev = None
            output_i = []
            h_t = None
            for t, (c_it, d_it, n_it, len_it) in enumerate(zip(code_x_i, divided_i, neighbor_i, range(len_i))):
                co_embeddings, no_embeddings = self.graph_layer(c_it, n_it, c_embeddings, n_embeddings)
                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
                output_i.append(output_it)
            output_i = self.attention(torch.vstack(output_i))
            output.append(output_i)
        output = torch.vstack(output)
        output = self.classifier(output)
        return output
```

Results

In this section, you should finish training your model training or loading your trained model. That is a great experiment! You should share the results with others with necessary metrics and figures.

Please test and report results for all experiments that you run with:

- specific numbers (accuracy, AUC, RMSE, etc)
- figures (loss shrinkage, outputs from GAN, annotation or label of sample pictures, etc)

In [8]:
```python
from utils import load_adj, EHRDataset, format_time, MultiStepLRScheduler
from metrics import evaluate_codes, evaluate_hf
import time
```

In [9]:
```python
# Pulled these hyperparameters from source code (train.py)
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 = 1  # 200  # decreased this to just 10 for testing purposes

seed = 6669
random.seed(seed)
np.random.seed(seed)
torch.manual_seed(seed)
torch.cuda.manual_seed(seed)

use_cuda = False
device = torch.device('cuda' if torch.cuda.is_available() and use_cuda else 'cpu')
```

In [10]:
```python
# Loading the data
dataset = 'mimic3'
task = 'h'

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)
```

```
loading train data ...
loading valid data ...
loading test data ...
```

In [11]:
```python
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

task_conf = {
    'm': {
        'dropout': 0.45,
        'output_size': code_num,
        'evaluate_fn': evaluate_codes,
        'lr': {
            'init_lr': 0.01,
            'milestones': [20, 30],
            'lrs': [1e-3, 1e-5]
        }
    },
    'h': {
        'dropout': 0.0,
        'output_size': 1,
        'evaluate_fn': evaluate_hf,
        'lr': {
            'init_lr': 0.01,
            'milestones': [1, 2],
            'lrs': [1e-3, 1e-4, 1e-5]
        }
    }
}
```

In [12]:
```python
# Training the model
test_historical = historical_hot(valid_data.code_x, code_num, valid_data.visit_lens)

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)
if not os.path.exists(param_path):
    os.makedirs(param_path)

model = 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)
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)

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))
    valid_loss, f1_score = evaluate_fn(model, valid_data, loss_fn, output_size, test_historical)
    torch.save(model.state_dict(), os.path.join(param_path, '%d.pt' % epoch))
```

```
721085
Epoch 1 / 1:
    Step 188 / 188, time cost: 5m11.6s, loss: 0.516161484
    Evaluation: loss: 0.4826 --- auc: 0.8402 --- f1_score: 0.7253
```

Model comparison

In [13]:
```python
# I have included the training results in the `results/` directory. I did not yet run the baseline model to make comparisons.
```

Discussion

In this section,you should discuss your work and make future plan. The discussion should address the following questions:

- Make assessment that the paper is reproducible or not.
- Explain why it is not reproducible if your results are kind negative.
- Describe "What was easy" and "What was difficult" during the reproduction.
- Make suggestions to the author or other reproducers on how to improve the reproducibility.
- What will you do in next phase.

The paper was reproducible. For this, I was able to download data MIMIC-III and MIMIC-IV data from https://physionet.org/. And then for the rest, from data processing to model building to model training and metrics evaluation, I was able to use the original papers code from https://github.com/LuChang-CS/Chet/tree/master. After looking at the performance of the model against the test dataset, the results align with the those from the paper for the heart failure prediction task. The inital runs were based on the replicated Chet model with just 5 epoches instead of the 200 epoches specified in the paper because of the runtime limitations. It was really easy to replicate the whole process mentioned in the paper because they provided the source code. But it was a little difficult to get the data. I initally used the MIMIC-III demo dataset which ended up not giving any results as the dataset was too small. So it is highly suggested to get access to the full dataset before reproducing the Chet model. and also the model takes a long time to run so it was difficult to run with 200 epoches so for this initial testing purposes I only ran 5 epoches but I will eventually run the full 200 and compare the data. As part of the next phase, I will start to I will continue to test this model on MIMIC-IV data as for now I only ran it on MIMIC-III dataset. Also I have not yet tested the diagnosis prediction task so I will test the model for this task with both the datasets. The other thing that is mentioned in the paper is the baseline model which I have not built and tested with, so I can not confirm if the Chet model outperfomes the baseline models like the paper claims.

References

1. Lu, C., Han, T., & Ning, Y. (2022). Context-Aware Health Event Prediction via Transition Functions on Dynamic Disease Graphs. Proceedings of the AAAI Conference on Artificial Intelligence, 36(4), 4567-4574. https://doi.org/10.1609/aaai.v36i4.20380
2. Johnson, Alistair, et al. "MIMIC-IV" (version 2.2). PhysioNet (2023), https://doi.org/10.13026/6mm1-ek67.
3. Johnson, A., Pollard, T., & Mark, R. (2019). MIMIC-III Clinical Database Demo (version 1.4). PhysioNet. https://doi.org/10.13026/C2HM2Q.