final_team_126

May 7, 2024

1 Project Final

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

2 Introduction

This project's goal is to reproduce the model proposed in the *Context-aware Health Event Prediction via Transition Functions on Dynamic Disease Graphs* (Lu et al. 2022) paper. The paper discusses that the current deep learning models on disease classification and prediction using longitudinal Electronic Health Record (EHR) data treat disease diagnoses as independent events within individual visits. However, in the real-world diseases and symptoms can be interlinked and can reflect hidden 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.

3 Github Repository

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

4 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.

5 Methodology

5.0.1 Environment

- Python Version = 3.9.16
- All the necssary python packages are listed in `requirements.txt` file and you can download them by running the command:

  ```
  pip install -r requirements.txt
  ```

5.0.2 Data

Source of the data: Currently I am using **MIMIC-III Clinical Database** and **MIMIC-IV Clinical Database**. This data is coming from https://physionet.org/content/mimiciii/1.4/ and https://physionet.org/content/mimiciv/2.2/. The necessary zip files are already downloaded into `data/download/` directory.

Data process:

1. Inital Data setup: From both the datasets, we just need `admissions.csv`, `diagnosis_icd.csv`, and `patients.csv` data files. I have the download zip files of these located under `data/download/`. And then `inital_data_setup.sh` script can be used to unzip these files into `data/mimic3/raw` and `data/mimic4/raw` directories.
2. Then follow the same data processing steps as in the paper. I have pulled the original scripts for preprocessing from papers repo into the `prepocess` directory. And I am running the `run_preprocess.py` script (copied into the next cell) on the dataset.

Statistics: The output of the followin script prints out the necessary statistics. 1. MIMIC-III data: There are 7493 patients in total from 2001 to 2012 with max visits of 42 and average visits of 2.6 per patient. The average diagnosis code per visit is 13.06. Using the same configurations as the original paper, we have randomly split the data into training, validation and test set of size 6000, 493 and 1000 respectively. 2. MIMIC-IV data: There are 10000 patients in total from 2013 to 2019 with max visits of 93 and average visits of 3.79 per patient. The average diagnosis code per visit is 13.51 diagnose codes per visit. Using the same configurations as the original paper, we have randomly split the data into training, validation and test set of size 8000, 1000 and 1000 respectively.

Overview of key statistics:

Dataset	MIMIC-III	MIMIC-IV
# patients	7,493	10,000
Max. # visit	42	55
Avg. # visit	2.66	3.66
# codes	4,880	6,102
Max. # codes per visit	39	50
Avg. # codes per visit	13.06	13.38

Table 1: Statistics of MIMIC-III and MIMIC-IV datasets

[1]:
```
# 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
        }
    }
    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)
```

[2]:
```
run_preprocess_modified(dataset='mimic3', from_saved=False)
```

```
parsing the csv file of admission …
        58976 in 58976 rows
parsing csv file of diagnosis …
        651047 in 651047 rows
calibrating patients by admission …
calibrating admission by patients …
saving parsed data …
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
```

[5]:
```
run_preprocess_modified(dataset='mimic4', from_saved=False)
```

```
loading ICD-10 to ICD-9 map …
loading patients anchor year …
parsing the csv file of admission …
        selecting valid admission …
                431231 in 431231 rows
                remaining 221815 rows
        221815 in 221815 rows
parsing csv file of diagnosis …
        mapping ICD-10 to ICD-9 …
                4756326 in 4756326 rows
        4756326 in 4756326 rows
calibrating patients by admission …
calibrating admission by patients …
saving parsed data …
patient num: 10000
max admission num: 93
mean admission num: 3.79
max code num in an admission: 39
mean code num in an admission: 13.51
encoding code …
There are 5985 codes
generating code levels …
        100%00%
There are 8000 train, 1000 valid, 1000 test samples
generating code code adjacent matrix …
        8000 / 8000
building train codes features and labels …
        8000 / 8000
building valid codes features and labels …
        1000 / 1000
building test codes features and labels …
        1000 / 1000
generating train neighbors …
        8000 / 8000
generating valid neighbors …
        1000 / 1000
```

```
generating test neighbors …
        1000 / 1000
generating train middles …
        8000 / 8000
generating valid middles …
        1000 / 1000
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
```

5.0.3 Model

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

1. Citation to paper: 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. Citation to paper's repo: Lu, Chang. "LuChang-CS/Chet." GitHub, 16 Dec. 2023, github.com/LuChang-CS/Chet. Accessed 8 May 2024.
    - link to repo: https://github.com/LuChang-CS/Chet

Model Overview The Chet model can be made up of three layers: graph layer, transition layer and embedding layer. I will explain each of the layers below along with their implementation code.

[4]:
```
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. This is acchieved by the following memory-efficient calculations: - `ZDt = m^t (M +A(m^t M) + A(n^t N))` –> aggregated diagnosis local and global context - `ZNt = n^t (N +A(n^t N) + A(m^t M))` –> aggregated neighbor global context

where M,N represent embedding matrices of diseases for diagnoses and neighbors respectively, A is the static adjacency matrix, m^t and n^t represent diagnoses and neighbors code in t visit.

The GNN outputs are calculated with a fully connected layer using LeakyReLU as the activation function: - `Ht = LeakyReLU(Zt * W) R^(d×s)`

[5]:
```
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 to learn the disease development schemes.

This layer takes `m^t` (vector of diagnosis codes) and divided it into three disjoint parts to represent longstanding and new-onset diseases: 1. Persistent diseases: represents diagnoses in visit `t` that are also diagnoses in visit `t - 1` 2. Emerging neighbors: represents diagnoses in visit `t` that are neighbors in visit `t - 1` 3. Emerging unrelated diseases: represents diagnoses in visit `t` that are unrelated diseases in visit `t - 1` Using these three parts of `m^t` with three transition functions for each part, historical contect is extracted from previous visits.

Then a modified gated recurrent unit (M-GRU) is used to model continuous features, that is to calculate the hidden values of `m^tp` (persistent diseases) to output `h^tp`

Lastly, max pooling is applied to the transistion output of the three partitoins to calculate the visit embedding `v^t`.

[6]:
```
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 The embedding layer applies a location-based attention to calculate the final hidden representation of all visits embeddings. And the patient embedding `o` will be used in a classifier for final predictions of a specific task.

[7]:
```
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

[8]:
```
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
```

6 Training

Hyperparams Most of the hyperparameters are kept the same as from the original paper: - hidden_size = 150 - batch_size = 32 - dropout_rate = 0.45 (for diagnosis prediction) - dropout_rate = 0.0 (for heart failure prediction)

The only parameter that have been modified are epochs and learning rate. I decreased the epochs from 200 to 20 and 10. And adjusted the learing rates accordingly with inital one starting at 0.01 for epoch 0.01: - lrs: [1e-3, 1e-5] (for diagnosis prediction) - lrs: [1e-3, 1e-4, 1e-5] (for heart failure prediction)

6.0.1 Computational requirements

1. All of the code is in python and uses pytorch packages.
2. Model was ran on my local computer with the specs: 32GB memory, Apple M2 Pro chip, 12 CPUs.
   - For training both mimic3 and mimic4 datasets for both diagnosis and heart failure tasks:
     - It took about 30 minutes for each epoch
     - Ran it for 20 epochs for a total run time of about 10 hours
3. As test to check performace of GPU, I ran it in Google Colab with their standard GPUs on just mimic3 data:
   - took about 6 minutes per epoch
   - Ran for 20 epochs for total run time of about 1 hour So clearly the use of GPUs will help improve runtime significantly compared to running on CPUs.

[9]:
```
from utils import load_adj, EHRDataset, format_time, MultiStepLRScheduler
from metrics import evaluate_codes, evaluate_hf
import time
```

[14]:
```
# 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 = 20  # 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')
```

[24]:
```
# Loading the data
dataset = 'mimic3'
task = 'm'

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 …
```

[25]:
```
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': [15, 17],
            'lrs': [1e-3, 1e-5]
```

```
        }
    },
    '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]
        }
    }
}
```

[ ]:
```
# 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))
```

6.1 Results

I have the results under `results/` in text files.

The models were evaluated against the test datasets and the results are coming from training each of the Chet models with 20 epochs instead of 200 epochs (as suggested in the paper). The performace of the model did align with the data presented in the paper for both the diagnosis and heart failure tasks.

For heart failure: - MIMIC-III: - Paper results: - F1 score: 73.08 - AUC: 86.14 - My results: - F1 score: 72.00 - AUC: 85.56 - MIMIC-IV: - Paper results: - F1 score: 71.14 - AUC: 90.83 - My results: - F1 score: 74.45 - AUC: 93.25 As we can see, my results in terms of F1 score and AUC are very close to those presented in the paper even though these are based on just 20 epochs.

7 Discussion

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 and diagnosis prediction task despite training the model on just 5 epoches instance of 200 recommended by the paper.

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 I highly suggest getting access to the full dataset before reproducing the Chet model. Also the model takes a long time to run so it was difficult to run with 200 epoches so instead I only ran 20 epoches. But even with this difference the model results were were compariable to the data presented in the paper.

What was easy: 1. MIMIC data is easily avaliable as long as you complete the process to get access to it 2. The original code base is easy to follow and reproduce.

What was difficult: 1. The biggest difficult is training the model due to computational requirements. On cpus, the models run for hours where as with GPUs it is significantly faster but avaliability and cost of GPUs is limiting. 2. The baseline model (CGL) was hard to repoduce and I was never able to get it to run. It seems to require some manual adjustments with the data processing as well as with the models adjacency graph.

Recommendations 1. It would be nice if there were more comments within the code for better readability. Also providing some refrences within the code to the equations in the paper would help with understanding the translation between the paper and code. 2. I think mode details on data preprocessing would be helpful in understanding how the data is actually be prepped for the models. I blindly just used the provided code but it would be good to get a better understanding of what is being done to the data. 3. Providing a small set of data with different hyperparamters for testing the model would be nice. Due to the computational requirements it is hard to test the model. So having that small dataset would help people play around with the model a bit.

8 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.
4. Lu, Chang. "LuChang-CS/Chet." GitHub, 16 Dec. 2023, github.com/LuChang-CS/Chet. Accessed 8 May 2024.