DL4H_Team_88_Final.ipynb
File Edit View Insert Runtime Tools Help All changes saved
Comment Share
+ Code + Text
Connect

TransformEHR: transformer-based encoder-decoder generative model to enhance prediction of disease outcomes using electronic health records.

CS598 Project

Anikesh Haran - anikesh2@illinois.edu
Satvik Kulkarni - satvikk2@illinois.edu
Changhua Zhan - zhan36@illinois.edu

Video Link

https://mediaspace.illinois.edu/media/t/1_dih6uxuo

Introduction

The paper addresses the pressing need for accurate prediction of clinical diseases and outcomes using electronic health records (EHRs). Specifically, it focuses on the problem of disease prediction and outcome forecasting, which holds immense significance in enhancing patient care and healthcare management. This problem involves intricate feature engineering and data processing due to the complexity and interrelation of various diseases and outcomes. Additionally, the challenge lies in achieving high predictive accuracy amidst the vast and heterogeneous nature of EHR data. Traditional machine learning methods have been employed but are being outperformed by deep learning techniques.

State of the Art Methods

The paper introduces TransformEHR, a novel denoising sequence to sequence transformer model, which tackles the limitations of existing methods. It innovatively pretrains on longitudinal EHRs to predict complete sets of ICD codes for future visits. The method's innovation lies in its generative encoder-decoder framework, which incorporates self-attention and cross-attention mechanisms. TransformEHR surpasses state-of-the-art BERT models, particularly excelling in predicting uncommon ICD codes.

TransformEHR

The paper presents TransformEHR as a solution to the challenges in disease prediction and outcome forecasting. Its key innovation is the novel pretraining objective, which predicts all diseases and outcomes for future visits using longitudinal EHR data. Additionally, its generative encoder-decoder framework outperforms existing encoder-based models due to its attention mechanisms. TransformEHR achieves significant improvements in predicting both common and uncommon ICD codes, showcasing its effectiveness.

General Problem

Accurately predicting disease outcomes using electronic health records (EHRs) is crucial for preventative medicine, personalized healthcare and treatment planning. Traditional machine learning methods have achieved success in this domain, but recent advancements in deep learning offer the potential for even greater accuracy by harnessing the complexity and temporal dynamics of EHR data to enhance disease outcome prediction.

Specific Approach

This proposal aims to implement and evaluate the TransformEHR model, a transformer-based encoder-decoder generative model specifically designed for disease outcome prediction using EHRs. The model will be trained on a large dataset of anonymized patient records to learn meaningful representations and patterns associated with disease progression and outcomes.

Contribution to Research Regime

The paper's contributions are multifaceted. Firstly, it proposes a new pretraining objective that captures complex interrelations among diseases and outcomes, addressing a critical gap in existing methods. Secondly, its innovative encoder-decoder framework sets a new standard for predictive modeling using EHRs, achieving superior performance compared to state-of-the-art methods. Thirdly, the study demonstrates the potential of TransformEHR in clinical screening and intervention, highlighting its practical significance. Overall, the paper significantly advances the field by offering a robust and effective solution to disease prediction and outcome forecasting using EHR data.

Scope of Reproducibility:

The reproducibility scope entails implementing and evaluating the TransformEHR model, a transformer-based encoder-decoder generative model specifically designed for disease outcome prediction using Electronic Health Records (EHRs). The model will undergo training on the MIMIC-IV dataset, consisting of deidentified patient records. The objective is to validate the model's capacity to learn meaningful representations and patterns associated with disease progression and outcomes using the provided dataset.

Hypotheses

- TransformEHR will achieve competitive performance compared to traditional machine learning models in predicting various disease outcomes using EHR data.
- The pre-training objective employed in TransformEHR, specifically predicting all future diagnoses, will improve the model's generalizability to diverse clinical prediction tasks.
- The model will effectively capture temporal dependencies and complex patterns within EHR data, leading to more accurate predictions.
- We will strive to distill complex patterns learned by TransformEHR into interpretable insights for clinicians, while achieving interpretability is inherently challenging in deep learning models.

Methodology

Pretrain-Finetune Paradigm

The Pretrain-Finetune paradigm is a widely used strategy in deep learning that involves two distinct phases to train a model effectively. In the pretraining phase, the model is trained on a large dataset using unsupervised or self-supervised learning tasks, such as language modeling or image reconstruction. This phase aims to capture general patterns and features from the data domain, leveraging the vast amount of information available in the large dataset. The pretrained model learns rich representations and general knowledge, which can be transferred to various downstream tasks.

Following pretraining, the finetuning phase involves adapting the pretrained model to a specific task or domain by fine-tuning its parameters using a smaller, domain-specific dataset with labeled data. This dataset is typically more focused on the target task, such as classification or sequence labeling. By finetuning on this dataset, the model refines its learned representations to better suit the nuances and intricacies of the specific task. The combination of pretraining on a large dataset and finetuning on a smaller task-specific dataset allows the model to leverage both general knowledge and task-specific information, leading to improved performance and robustness on the target task.

Transform EHR

Step #1 - first TransformEHR is pre-trained with a generative encoder-decoder transformer on a large set of EHR data. TransformEHR will learn the probability distribution of ICD codes against random distribution through the correlation of cross attention.

Step #2 - in the downstream finetuning, TransformEHR predicts a single disease or outcome. Through the calculated attention weights above, TransformEHR is able to identify top indicators for the predictions. This is shown in the picture below.

Environment

Platform & Notebook

We have used Google Cloud Platform [GCP] - **Colab Enterprise Vertex AI Runtime** to reproduce this paper.

Python Version

- 3.10.12

Packages & Dependencies

- Pytorch - 2.2.1+cu121
- Numpy - 1.25.2
- PyHealth - 1.1.6

Infrastrcture & Capacity

- Colab Enterprise Vertex AI Runtime
- Machine type - e2-highmem-16
- CPU - 16
- Memory - 128

```
[ ] #Install required packages
    !pip install pyhealth
```

```
Collecting pyhealth
  Downloading pyhealth-1.1.6-py2.py3-none-any.whl (311 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 311.6/311.6 kB 6.2 MB/s eta 0:00:00
Requirement already satisfied: torch>=1.8.0 in /usr/local/lib/python3.10/dist-packages (from pyhealth) (2.2.1+cu121)
Requirement already satisfied: torchvision>=0.9.0 in /usr/local/lib/python3.10/dist-packages (from pyhealth) (0.17.1+cu121)
Collecting rdkit>=2022.03.4 (from pyhealth)
  Downloading rdkit-2023.9.6-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (34.9 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 34.9/34.9 MB 36.9 MB/s eta 0:00:00
Requirement already satisfied: scikit-learn>=0.24.2 in /usr/local/lib/python3.10/dist-packages (from pyhealth) (1.2.2)
Requirement already satisfied: networkx>=2.6.3 in /usr/local/lib/python3.10/dist-packages (from pyhealth) (3.3)
Collecting pandas<2,>=1.3.2 (from pyhealth)
  Downloading pandas-1.5.3-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (12.1 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 12.1/12.1 MB 82.5 MB/s eta 0:00:00
Collecting pandarallel>=1.5.3 (from pyhealth)
  Downloading pandarallel-1.6.5.tar.gz (14 kB)
  Preparing metadata (setup.py) ... done
Collecting mne>=1.0.3 (from pyhealth)
  Downloading mne-1.7.0-py3-none-any.whl (7.4 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 7.4/7.4 MB 86.1 MB/s eta 0:00:00
Collecting urllib3<=1.26.15 (from pyhealth)
  Downloading urllib3-1.26.15-py2.py3-none-any.whl (140 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 140.9/140.9 kB 14.9 MB/s eta 0:00:00
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (from pyhealth) (1.25.2)
Requirement already satisfied: tqdm in /usr/local/lib/python3.10/dist-packages (from pyhealth) (4.66.2)
Requirement already satisfied: decorator in /usr/local/lib/python3.10/dist-packages (from mne>=1.0.3->pyhealth) (4.4.2)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/dist-packages (from mne>=1.0.3->pyhealth) (3.1.3)
Requirement already satisfied: lazy-loader>=0.3 in /usr/local/lib/python3.10/dist-packages (from mne>=1.0.3->pyhealth) (0.4)
Requirement already satisfied: matplotlib>=3.5.0 in /usr/local/lib/python3.10/dist-packages (from mne>=1.0.3->pyhealth) (3.7.1)
Requirement already satisfied: packaging in /usr/local/lib/python3.10/dist-packages (from mne>=1.0.3->pyhealth) (24.0)
Requirement already satisfied: pooch>=1.5 in /usr/local/lib/python3.10/dist-packages (from mne>=1.0.3->pyhealth) (1.8.1)
Requirement already satisfied: scipy>=1.7.1 in /usr/local/lib/python3.10/dist-packages (from mne>=1.0.3->pyhealth) (1.11.4)
Collecting dill>=0.3.1 (from pandarallel>=1.5.3->pyhealth)
  Downloading dill-0.3.8-py3-none-any.whl (116 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 116.3/116.3 kB 12.8 MB/s eta 0:00:00
Requirement already satisfied: psutil in /usr/local/lib/python3.10/dist-packages (from pandarallel>=1.5.3->pyhealth) (5.9.5)
Requirement already satisfied: python-dateutil>=2.8.1 in /usr/local/lib/python3.10/dist-packages (from pandas<2,>=1.3.2->pyhealth) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas<2,>=1.3.2->pyhealth) (2023.4)
Requirement already satisfied: Pillow in /usr/local/lib/python3.10/dist-packages (from rdkit>=2022.03.4->pyhealth) (9.4.0)
Requirement already satisfied: joblib>=1.1.1 in /usr/local/lib/python3.10/dist-packages (from scikit-learn>=0.24.2->pyhealth) (1.4.0)
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.10/dist-packages (from scikit-learn>=0.24.2->pyhealth) (3.5.0)
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-packages (from torch>=1.8.0->pyhealth) (3.14.0)
Requirement already satisfied: typing-extensions>=4.8.0 in /usr/local/lib/python3.10/dist-packages (from torch>=1.8.0->pyhealth) (4.11.0)
Requirement already satisfied: sympy in /usr/local/lib/python3.10/dist-packages (from torch>=1.8.0->pyhealth) (1.12)
Requirement already satisfied: fsspec in /usr/local/lib/python3.10/dist-packages (from torch>=1.8.0->pyhealth) (2023.6.0)
Collecting nvidia-cuda-nvrtc-cu12==12.1.105 (from torch>=1.8.0->pyhealth)
  Using cached nvidia_cuda_nvrtc_cu12-12.1.105-py3-none-manylinux1_x86_64.whl (23.7 MB)
Collecting nvidia-cuda-runtime-cu12==12.1.105 (from torch>=1.8.0->pyhealth)
  Using cached nvidia_cuda_runtime_cu12-12.1.105-py3-none-manylinux1_x86_64.whl (823 kB)
Collecting nvidia-cuda-cupti-cu12==12.1.105 (from torch>=1.8.0->pyhealth)
  Using cached nvidia_cuda_cupti_cu12-12.1.105-py3-none-manylinux1_x86_64.whl (14.1 MB)
Collecting nvidia-cudnn-cu12==8.9.2.26 (from torch>=1.8.0->pyhealth)
```

```
  Using cached nvidia_cudnn_cu12-8.9.2.26-py3-none-manylinux1_x86_64.whl (731.7 MB)
Collecting nvidia-cublas-cu12==12.1.3.1 (from torch>=1.8.0->pyhealth)
  Using cached nvidia_cublas_cu12-12.1.3.1-py3-none-manylinux1_x86_64.whl (410.6 MB)
Collecting nvidia-cufft-cu12==11.0.2.54 (from torch>=1.8.0->pyhealth)
  Using cached nvidia_cufft_cu12-11.0.2.54-py3-none-manylinux1_x86_64.whl (121.6 MB)
Collecting nvidia-curand-cu12==10.3.2.106 (from torch>=1.8.0->pyhealth)
  Using cached nvidia_curand_cu12-10.3.2.106-py3-none-manylinux1_x86_64.whl (56.5 MB)
```

```
[ ] # import  packages
    import numpy as np
    import torch
    import torch.nn as nn
    import torch.nn.functional as F
    import pickle
    from torch.utils.data import Dataset
    from torch.utils.data import DataLoader
    import math
    import torch.nn.utils.rnn as rnn_utils
```

Data

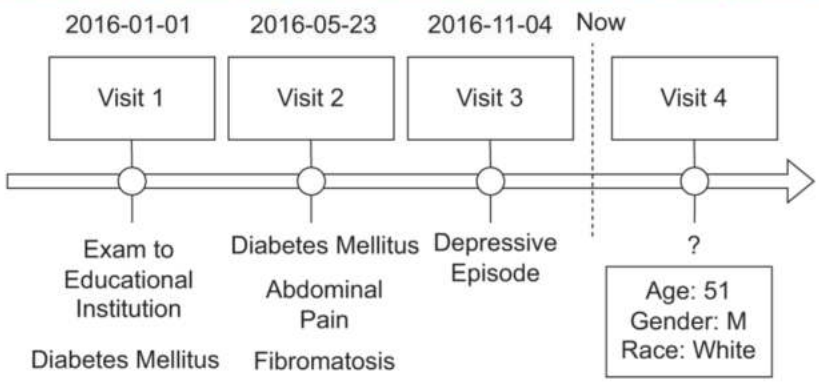
The dataset we plan to use in this project is MIMIC-IV from https://physionet.org. The MIMIC-IV dataset includes intensive care unit patients admitted to the Beth Israel Deaconess Medical Center in Boston, Massachusetts, comprises deidentified patient records used for medical research and analysis. It encompasses a wide range of clinical data, including demographic information, vital signs, laboratory results, medications, procedures, and clinical notes. MIMIC-IV offers longitudinal Electronic Health Records (EHRs) from various healthcare facilities, providing a comprehensive view of patient health trajectories. This dataset serves as a valuable resource for studying disease progression, treatment outcomes, predictive modeling, and other healthcare-related research endeavors.

Since the dataset contains information from 2008 to 2019 but the implementation of ICD-10CM started from October 2015, to mimic the same dataset as per the paper, we have converted ICD9CM codes into ICD10CM codes first to have enough patients and visits for the cohorts for pretraining, resulting in a dataset of 180733 patients.

Longitudnal EHR Data

Fig. 1: Exemplar EHR sequence of a patient (white, male, age 51).

From: TransformEHR: transformer-based encoder-decoder generative model to enhance prediction of disease outcomes using electronic health records



This EHR contains demographic and ICD codes from 3 previous visits, including Z-codes such as exam to educational institution Z02.0. The pretraining objective is to predict all diseases and outcomes in the next visit. All these values are artificial and for illustration purposes.

Data Includes

- Raw Data - MIMIC IV tables
  - Admissions,
  - Patient and
  - Icd_diagonosis codes
- Descriptive Statistics
  - Dataset: MIMIC4Dataset
  - Number of patients: 180733
  - Number of visits: 431231
  - Number of visits per patient: 2.3860
  - Number of events per visit in diagnoses_icd: 11.0296
    - Train and Valdiation set - TBD

Data Processing (feature engineering)

MIMIC-IV Cohort

Our pretraining cohort comprises 180733 patients and 431231 admissions. As per the paper To evaluate pretrained models, we created two disease/outcome agnostic prediction (DOAP) datasets—one for common and one for uncommon diseases/outcomes. We selected 10 ICD-10CM codes with the highest prevalence (prevalence ratio >2%) in our pretraining cohort for our common disease/outcome DOAP dataset. As for the set of uncommon diseases/outcomes, we followed the FDA guidelines30 to randomly select 10 ICD-10CM codes with a prevalence ratio ranging from 0.04% to 0.05% in our pretraining cohort. The lists of common and uncommon diseases/outcomes are shown in Table 1.

Data Processing

For data pre-processing we have used PyHealth pyhealth.datasets.MIMIC4Dataset to process the unstructured raw data into a structured dataset object. See the implementation section below.

Created Common & Uncommon DataSet Extract Relevant Information: Extract the necessary information from the MIMIC-IV dataset, including patient records, diagnoses, and outcomes.

- Identify Prevalent ICD-10CM Codes: Identify the prevalent ICD-10CM codes in the pretraining cohort. For the common disease/outcome DOAP dataset, select 10 ICD-10CM codes with the highest prevalence ratio (>2%) in the pretraining cohort.
- Select Uncommon ICD-10CM Codes: Follow the FDA guidelines to randomly select 10 ICD-10CM codes with a prevalence ratio ranging from 0.04% to 0.05% in the pretraining cohort for the set of uncommon diseases/outcomes.
- Create Common Disease/Outcome DOAP Dataset: Filter the patient records to include only those with the selected common ICD-10CM codes. This will form the common disease/outcome DOAP dataset.
- Create Uncommon Disease/Outcome DOAP Dataset: Similarly, filter the patient records to include only those with the selected uncommon ICD-10CM codes. This will form the uncommon disease/outcome DOAP dataset.

index	ICD-10-CM Code	Description
0	I10	Essential (primary) hypertension
1	E785	Hyperlipidemia, unspecified
2	Z87891	Personal history of nicotine dependence
3	K219	Gastro-esophageal reflux disease without esophagitis
4	F329	Major depressive disorder, unspecified
5	I2510	Atherosclerotic heart disease of native coronary artery without angina pectoris
6	F419	Unspecified anxiety disorder
7	N179	Chronic kidney disease, unspecified
8	Z794	Long-term (current) use of insulin
9	Z7901	Long-term (current) use of opiate analgesic

index	ICD-10-CM Code	Description
0	N94.6	Dyspareunia, unspecified
1	T47.1X5D	Poisoning by antineoplastic and immunosuppressive drugs, accidental (unintentional), subsequent encounter
2	O30.033	Triplet pregnancy, fetus 3
3	I70234	Atherosclerosis of native arteries of extremities with gangrene, bilateral legs
4	I95.2	Hypotension, unspecified
5	Z34.83	Supervision of high-risk pregnancy with other poor reproductive or obstetric history
6	C8518	Diffuse large B-cell lymphoma, lymph nodes of axilla and upper limb
7	L89.891	Pressure ulcer of other site, stage 1
8	D126	Benign neoplasm of colon
9	I201	Unstable angina

Show 25 per page

Load Data

The MIMIC-IV dataset, a valuable resource for healthcare research, comes with stringent data sharing restrictions designed to protect patient privacy and ensure ethical use. Access to MIMIC-IV necessitates signing a Data Use Agreement (DUA) with the MIT Laboratory for Computational Physiology, outlining terms such as authorized use, privacy protection measures, and attribution requirements.

Since we are bound to not share the RAW data. We have pre-processed the raw data and created the pickle files for quick loading and model traninign. We have checked in the processed pickle files into GitHub under data folder.

RAW Data Processing

For initial raw data processing, we have used PyHealth MIMIC4Dataset

```
[ ] """from pyhealth.datasets import MIMIC4Dataset

    # dir and function to load raw data
    root = '/content/drive/MyDrive/DLH/MIMIC4/CSV/'

    def load_raw_data(raw_data_dir):
      # implement this function to load raw data to dataframe/numpy array/tensor
      mimic4_ds = MIMIC4Dataset(
        # Argument 1: It specifies the data folder root.
        root=raw_data_dir,

        # Argument 2: The users need to input a list of raw table names (e.g., DIAGNOSES_ICD.csv, PROCEDURES_ICD.csv).
        tables=["diagnoses_icd"],
        # Argument 3: This argument input a dictionary (key is the source code
        # vocabulary and value is the target code vocabulary .
        # Default is empty dict, which means the original code will be used.
        # We will use ICD10 codes.
        code_mapping={}
        )
      return mimic4_ds

    mimic4_ds = load_raw_data(root)

    mimic4_ds.info()"""
```

```
[ ] """# Statistics of the entire dataset.
    mimic4_ds.stat()

    # You can find the list of all available tables in this dataset as
```

```
mimic4_ds.available_tables"""
```

```
Statistics of base dataset (dev=False):
        - Dataset: MIMIC4Dataset
        - Number of patients: 180733
        - Number of visits: 431231
        - Number of visits per patient: 2.3860
        - Number of events per visit in diagnoses_icd: 11.0296

['diagnoses_icd']
```

```
[ ] """#Save data object to drive for quick retrival
    import pickle

    # Assuming your data object is named 'data_object'
    mimic4_ds_object_path = '/content/drive/MyDrive/DLH/MIMIC4/PKL/mimic4_ds.pkl'

    # Save the data object to Google Drive
    with open(mimic4_ds_object_path, 'wb') as f:
        pickle.dump(mimic4_ds, f)"""
```

```
[ ] #Load MIMIC4 data from google drive
    import pickle

    # Path to the saved data object
    data_object_path = '/content/drive/MyDrive/DLH/MIMIC4/PKL/mimic4_ds.pkl'

    # Load the data object from Google Drive
    with open(data_object_path, 'rb') as f:
        mimic4_data = pickle.load(f)

    # Statistics of the entire dataset.
    mimic4_data.stat()

    # You can find the list of all available tables in this dataset as
    mimic4_data.available_tables
```

```
Statistics of base dataset (dev=False):
        - Dataset: MIMIC4Dataset
        - Number of patients: 180733
        - Number of visits: 431231
        - Number of visits per patient: 2.3860
        - Number of events per visit in diagnoses_icd: 11.0296

['diagnoses_icd']
```

Sample Data

```
[ ] """# get patient dictionary
    patient_dict = mimic4_data.patients
    print(list(patient_dict.keys())[:10])

    # get the "10000032" patient
    patient = patient_dict["10000032"]
    print(patient)"""
```

```
                        - Table: diagnoses_icd
                        - Vocabulary: ICD9CM
                        - Timestamp: None
                - Event from patient 10000032 visit 29079034:
                        - Code: 7994
                        - Table: diagnoses_icd
                        - Vocabulary: ICD9CM
                        - Timestamp: None
                - Event from patient 10000032 visit 29079034:
                        - Code: 2761
                        - Table: diagnoses_icd
                        - Vocabulary: ICD9CM
                        - Timestamp: None
                - Event from patient 10000032 visit 29079034:
                        - Code: 78959
                        - Table: diagnoses_icd
                        - Vocabulary: ICD9CM
                        - Timestamp: None
                - Event from patient 10000032 visit 29079034:
                        - Code: 2767
                        - Table: diagnoses_icd
                        - Vocabulary: ICD9CM
                        - Timestamp: None
                - Event from patient 10000032 visit 29079034:
                        - Code: 3051
                        - Table: diagnoses_icd
                        - Vocabulary: ICD9CM
                        - Timestamp: None
                - Event from patient 10000032 visit 29079034:
                        - Code: V08
                        - Table: diagnoses_icd
                        - Vocabulary: ICD9CM
                        - Timestamp: None
                - Event from patient 10000032 visit 29079034:
                        - Code: V4986
                        - Table: diagnoses_icd
                        - Vocabulary: ICD9CM
                        - Timestamp: None
                - Event from patient 10000032 visit 29079034:
                        - Code: V462
                        - Table: diagnoses_icd
                        - Vocabulary: ICD9CM
                        - Timestamp: None
                - Event from patient 10000032 visit 29079034:
                        - Code: 496
                        - Table: diagnoses_icd
                        - Vocabulary: ICD9CM
                        - Timestamp: None
                - Event from patient 10000032 visit 29079034:
                        - Code: 29680
```

```
        - Table: diagnoses_icd
        - Vocabulary: ICD9CM
        - Timestamp: None
    - Event from patient 10000032 visit 29079034:
        - Code: 5715
        - Table: diagnoses_icd
        - Vocabulary: ICD9CM
        - Timestamp: None
```

Creating Common and Uncommon disease/outcome agnostic prediction (DOAP) datasets.

```
"""import random
from collections import Counter

# Step 1: Calculate the prevalence of each ICD-10CM code
icd_counter = Counter()

total_patients = 180733

for patient in mimic4_sample:
    for icd_code in patient['icd_codes']:
        icd_counter[icd_code] += 1

# Step 2: Select top 10 ICD-10CM codes with highest prevalence ratio (>2%) for common dataset
# common_icd_codes = [icd_code for icd_code, count in icd_counter.items() if (count / total_patients) > 0.02][:10]
# NOTE: we need diseases with the top 10 prevalence ratio
common_icd_codes = [icd_code for icd_code, count in icd_counter.most_common(10)]
# check whether all selected diseases has a prevalence ratio of > 2%
print(sum([count/total_patients > 0.02 for icd_code, count in icd_counter.items() if icd_code in common_icd_codes]))


# Step 3: Randomly select 10 ICD-10CM codes with prevalence ratio ranging from 0.04% to 0.05% for uncommon dataset
uncommon_icd_codes = [icd_code for icd_code, count in icd_counter.items() if 0.0004 <= (count / total_patients) <= 0.0005]
random.shuffle(uncommon_icd_codes)
uncommon_icd_codes = uncommon_icd_codes[:10]

# Step 4: Filter patient records to create common and uncommon datasets
common_disease_dataset = [patient for patient in mimic4_sample if any(icd in patient['icd_codes'] for icd in common_icd_codes)]
uncommon_disease_dataset = [patient for patient in mimic4_sample if any(icd in patient['icd_codes'] for icd in uncommon_icd_codes)]

# Print the selected ICD-10CM codes for common and uncommon datasets
print("Selected Common ICD-10CM Codes:", common_icd_codes)
print("Selected Uncommon ICD-10CM Codes:", uncommon_icd_codes)

# Optionally, print the lengths of the resulting datasets
print("Number of patients in Common Disease/Outcome DOAP Dataset:", len(common_disease_dataset))
print("Number of patients in Uncommon Disease/Outcome DOAP Dataset:", len(uncommon_disease_dataset))"""
```

The lists of common and uncommon diseases/outcomes are shown in Table 1 and Table 2 respectivly.

Table 1 - Common ICD-10CM Codes

```
[ ] import pandas as pd

    # Define the data for the table
    common_outcomes = {
        'ICD-10-CM Code': ['I10', 'E785', 'Z87891', 'K219', 'F329', 'I2510', 'F419', 'N179', 'Z794', 'Z7901'],
        'Description': [
            'Essential (primary) hypertension',
            'Hyperlipidemia, unspecified',
            'Personal history of nicotine dependence',
            'Gastro-esophageal reflux disease without esophagitis',
            'Major depressive disorder, unspecified',
            'Atherosclerotic heart disease of native coronary artery without angina pectoris',
            'Unspecified anxiety disorder',
            'Chronic kidney disease, unspecified',
            'Long-term (current) use of insulin',
            'Long-term (current) use of opiate analgesic'
        ]
    }

    # Create a DataFrame from the data
    common_outcomes_df = pd.DataFrame(common_outcomes)

    # Display the DataFrame
    common_outcomes_df
```

	ICD-10-CM Code	Description
0	I10	Essential (primary) hypertension
1	E785	Hyperlipidemia, unspecified
2	Z87891	Personal history of nicotine dependence
3	K219	Gastro-esophageal reflux disease without esoph...
4	F329	Major depressive disorder, unspecified
5	I2510	Atherosclerotic heart disease of native corona...
6	F419	Unspecified anxiety disorder
7	N179	Chronic kidney disease, unspecified
8	Z794	Long-term (current) use of insulin
9	Z7901	Long-term (current) use of opiate analgesic

Table 2 - Uncommon ICD-10CM Codes

```
[ ] import pandas as pd
```

```
# Define the data for the table
uncommon_outcomes = {
    'ICD-10-CM Code': ['N94.6', 'T47.1X5D', 'O30.033', 'I70234', 'I95.2', 'Z34.83', 'C8518', 'L89.891', 'D126', 'I201'],
    'Description': [
        'Dyspareunia, unspecified',
        'Poisoning by antineoplastic and immunosuppressive drugs, accidental (unintentional), subsequent encounter',
        'Triplet pregnancy, fetus 3',
        'Atherosclerosis of native arteries of extremities with gangrene, bilateral legs',
        'Hypotension, unspecified',
        'Supervision of high-risk pregnancy with other poor reproductive or obstetric history',
        'Diffuse large B-cell lymphoma, lymph nodes of axilla and upper limb',
        'Pressure ulcer of other site, stage 1',
        'Benign neoplasm of colon',
        'Unstable angina'
    ]
}

# Create a DataFrame from the additional data
uncommon_outcomes_df = pd.DataFrame(uncommon_outcomes)

# Display the DataFrame
uncommon_outcomes_df
```

	ICD-10-CM Code	Description
0	N94.6	Dyspareunia, unspecified
1	T47.1X5D	Poisoning by antineoplastic and immunosuppress...
2	O30.033	Triplet pregnancy, fetus 3
3	I70234	Atherosclerosis of native arteries of extremit...
4	I95.2	Hypotension, unspecified
5	Z34.83	Supervision of high-risk pregnancy with other ...
6	C8518	Diffuse large B-cell lymphoma, lymph nodes of ...
7	L89.891	Pressure ulcer of other site, stage 1
8	D126	Benign neoplasm of colon
9	I201	Unstable angina

Patient Age - PreProcessing

Current PyHealth based data processing does not compute age feature. hence we pre-processed the patient's age separately and created a pickle files for age feature for quick loading during model training.

```
[ ] """import csv

    # Define the path to the CSV file
    root = '/content/drive/MyDrive/DLH/MIMIC4/CSV/'
    patient_file_path = root + 'patients.csv'

    id2age = {}

    # read id and age from patients.csv and save it in a dictionary id2age
    def read_patient_age(file_path):
        with open(file_path, mode='r') as file:
            reader = csv.reader(file)
            next(reader)
            for row in reader:
                id2age[row[0]] = int(row[2])

    read_patient_age(patient_file_path)"""
```

Pre-Processing - transform_ehr_mimic4_fn

We have developed function **transform_ehr_mimic4_fn** to process individual patients and create feautres such as visit level details, icd codes and patinet;s demographic details such as age, gender and race.

To reduce the data complexity and need of high compute power, we have pre-processed the longitudnal EHR data and kept fixed length sequence of Visit & ICD-Codes.

- Visit Length - 4 visits per patient
- ICD Codes - 5 ICD codes per visit

Patients with less then 4 visits and less then 5 ICD diagnosis-codes have been discarded from the pre-traning cohort.

Total Patients In Pre-Training Cohort - 23206

```
[ ] """# Compute sequenced data for learning embeddings

    from datetime import datetime
    from pyhealth.medcode import CrossMap
    import random
    # set the random seed
    random.seed(0)

    # load the mapping from ICD9CM to CCSCM
    mapping_icd9cm_ccscm = CrossMap.load(source_vocabulary="ICD9CM", target_vocabulary="CCSCM")
    # load the mapping from CCSCM to ICD10CM
    mapping_ccscm_icd10cm = CrossMap.load(source_vocabulary="CCSCM", target_vocabulary="ICD10CM")

    #Calculate Patient's Age
    def calculate_age(birth_date, death_date):
      # Calculate age
      age = death_date.year - birth_date.year - ((death_date.month, death_date.day) < (birth_date.month, birth_date.day))
      return age

    types = {}
    gender2idx = {}
```

```
gender2idx = {}
race2idx = {}

def transform_ehr_mimic4_fn(patient):
    visit_idx = []
    newPatient = []
    age = 0
    gender = 0
    race = 0
    visit_dates = []
    #consider patient with 4 or more visits
    keep_patient = True
    if len(patient) >= 4:
      for i in range(len(patient)):
        #visit level details
        visit_idx.append(1 + i)

        visit = patient[i]
        conditions = []
        events = visit.get_event_list(table="diagnoses_icd")
        if(len(events) < 5):
          continue
        formatted_visit_date = visit.encounter_time.strftime("%Y-%m-%d")
        visit_dates.append(formatted_visit_date)

        for event in events:
          vocabulary = event.vocabulary
          code = ""
          if vocabulary == "ICD9CM":
            # map from ICD9CM to CCSCM
            ccscmCodes = mapping_icd9cm_ccscm.map(event.code)
            # in the case where one ICD9CM code maps to multiple CCSCM codes, randomly select one
            ccscmCode = random.choice(ccscmCodes)

            # map from CCSCM to ICD10CM
            icd10cmCodes = mapping_ccscm_icd10cm.map(ccscmCode)
            # in the case where one CCSCM code maps to multiple ICD10CM codes, randomly select one
            code = random.choice(icd10cmCodes)
          else:
            code = event.code

          if code in types:
            conditions.append(types[code])
          else:
            types[code] = len(types)
            conditions.append(types[code])

        # step 2: assemble the sample
        # if conditions is not empty, add the sample
        # if (conditions): # commented it out because len(visit_date) needs to be the same as len(newPatient)
        newPatient.append(conditions)

      if(len(conditions) >= 4):
        #visits.append(visit_idx)
        if len(newPatient) > 100:
          print(patient.patient_id,)
        #visit_dates.append(visit_date)
        #age.append(patient.anchor_age)

        # get age of patient using patient id and id2age dictionary
        #age.append(id2age[patient.patient_id])
        age = id2age[patient.patient_id]

        p_gender = patient.gender
        if p_gender in gender2idx:
          #gender.append(gender2idx[p_gender])
          gender = gender2idx[p_gender]
        else:
          gender2idx[p_gender] = len(gender2idx)
          #gender.append(gender2idx[p_gender])
          gender = gender2idx[p_gender]

        p_ethnicity = patient.ethnicity
        if p_ethnicity in race2idx:
          #race.append(race2idx[p_ethnicity])
          race = race2idx[p_ethnicity]
        else:
          race2idx[p_ethnicity] = len(race2idx)
          #race.append(race2idx[p_ethnicity])
          race = race2idx[p_ethnicity]
    return newPatient, visit_idx, age, gender, race, visit_dates"""
```

```
[ ] """# Pre-processing of MIMIC4 Data
    seqs = []
    all_visits = []
    all_age = []
    all_gender = []
    all_race = []
    all_visit_dates = []
    patient_dict = mimic4_data.patients
    for patient_id in mimic4_data.patients:
      patient = patient_dict[patient_id]
      seq, visit_numbers, age, gender, race, visit_dates = transform_ehr_mimic4_fn(patient)
      if seq and len(seq) >=4:
        seqs.append(seq)
        all_visits.append(visit_numbers)
        all_age.append(age)
        all_gender.append(gender)
        all_race.append(race)
        all_visit_dates.append(visit_dates)

    print(seqs[0])"""
    #[[0, 1, 2, 3, 4, 5, 6, 7], [8, 9, 10, 11, 12, 13, 14, 15], [16, 17, 18, 19, 20, 20, 4, 21, 22, 23], [24, 25, 26, 27, 28, 29, 30, 14, 31, 32, 4, 33, 34]]
```

Ordering Visits Based on Visit Dates

```
[ ] """# sort the visit_date and seqs based on the visit date
    from datetime import datetime
    sorted_seqs = []
    sorted_visit_dates = []
    for i in range(len(seqs)):
        visit_date = all_visit_dates[i]
        seq = seqs[i]
        visit_date_seq_tuple = [(visit_date[j], seq[j]) for j in range(len(seq))]
        visit_date_seq_tuple.sort(key=lambda x: datetime.strptime(x[0], "%Y-%m-%d"))

        sorted_visit_dates.append([x[0] for x in visit_date_seq_tuple])
        sorted_seqs.append([x[1] for x in visit_date_seq_tuple])

    seqs = sorted_seqs
    all_visit_dates = sorted_visit_dates
    print(seqs[0])
    print(all_visit_dates[0])"""

    #[[0, 1, 2, 3, 4, 5, 6, 7], [8, 9, 10, 11, 12, 13, 14, 15], [24, 25, 26, 27, 28, 29, 30, 14, 31, 32, 4, 33, 34], [16, 17, 18, 19, 20, 20, 4, 21, 22, 23]]
    #['2180-05-06', '2180-06-26', '2180-07-23', '2180-08-05']
```

```
[ ] """print(len(seqs))
    print(len(all_visits))
    print(len(all_visit_dates))
    print(len(all_gender))
    print(len(all_race))
    print(len(all_age))
    print(len(types))"""

    #23206
    #23206
    #23206
    #23206
    #23206
    #23206
    #51730
```

Create Pickle

In below code section we have created pickle files for all the data features - sequences, visit dates, gender , race & age and stored into filesystem.

Note - change the "path" according to your environment.

```
[ ] """import pickle

    mimic4_ds_seqs_path = '/content/seqs.pkl'
    mimic4_ds_visits_path = '/content/visits.pkl'
    mimic4_ds_visit_dates_path = '/content/dates.pkl'
    mimic4_ds_type_path = '/content/type.pkl'
    mimic4_ds_gender_path = '/content/gender.pkl'
    mimic4_ds_race_path = '/content/race.pkl'
    mimic4_ds_age_path = '/content/age.pkl'

    # Save the data object to Google Drive
    with open(mimic4_ds_seqs_path, 'wb') as f:
        pickle.dump(seqs, f)

    with open(mimic4_ds_visits_path, 'wb') as f:
        pickle.dump(all_visits, f)

    with open(mimic4_ds_visit_dates_path, 'wb') as f:
        pickle.dump(all_visit_dates, f)

    with open(mimic4_ds_type_path, 'wb') as f:
        pickle.dump(types, f)

    with open(mimic4_ds_gender_path, 'wb') as f:
        pickle.dump(all_gender, f)

    with open(mimic4_ds_race_path, 'wb') as f:
        pickle.dump(all_race, f)

    with open(mimic4_ds_age_path, 'wb') as f:
        pickle.dump(all_age, f)"""
```

Load Data for Model Training

```
[ ] #Load preprocessed data from google drive

    from google.colab import drive
    drive.mount('/content/drive', force_remount=True)
```

```
Mounted at /content/drive
```

```
[ ] #Load MIMIC4 data from google drive
    import pickle

    # Path to the saved data object
    """mimic4_ds_seqs_path = '/content/seqs.pkl'
    mimic4_ds_visits_path = '/content/visits.pkl'
    mimic4_ds_visit_dates_path = '/content/dates.pkl'
    mimic4_ds_type_path = '/content/type.pkl'
    mimic4_ds_gender_path = '/content/gender.pkl'
    mimic4_ds_race_path = '/content/race.pkl'
    mimic4_ds_age_path = '/content/age.pkl'"""
```

```
mimic4_ds_seqs_path = '/content/drive/MyDrive/DLH/MIMIC4/PKL/v3/seqs.pkl'
mimic4_ds_visits_path = '/content/drive/MyDrive/DLH/MIMIC4/PKL/v3/visits.pkl'
mimic4_ds_visit_dates_path = '/content/drive/MyDrive/DLH/MIMIC4/PKL/v3/dates.pkl'
mimic4_ds_type_path = '/content/drive/MyDrive/DLH/MIMIC4/PKL/v3/type.pkl'
mimic4_ds_gender_path = '/content/drive/MyDrive/DLH/MIMIC4/PKL/v3/gender.pkl'
mimic4_ds_race_path = '/content/drive/MyDrive/DLH/MIMIC4/PKL/v3/race.pkl'
mimic4_ds_age_path = '/content/drive/MyDrive/DLH/MIMIC4/PKL/v3/age.pkl'

# Load the data object from Google Drive
with open(mimic4_ds_seqs_path, 'rb') as f:
    seqs = pickle.load(f)

with open(mimic4_ds_visits_path, 'rb') as f:
    visits = pickle.load(f)

# Load the data object from Google Drive
with open(mimic4_ds_visit_dates_path, 'rb') as f:
    visit_dates = pickle.load(f)

# Load the data object from Google Drive
with open(mimic4_ds_type_path, 'rb') as f:
    icd_codes_types = pickle.load(f)

# Load the data object from Google Drive
with open(mimic4_ds_gender_path, 'rb') as f:
    gender = pickle.load(f)

# Load the data object from Google Drive
with open(mimic4_ds_race_path, 'rb') as f:
    race = pickle.load(f)

# Load the data object from Google Drive
with open(mimic4_ds_age_path, 'rb') as f:
    age = pickle.load(f)
```

Build The Dataset

First, we have implemented a custom dataset using PyTorch Dataset class, which will characterize the key features of the dataset we want to generate.

We will use the sequences of diagnosis-codes, gender, age, race and visit-dates as input for pretraning.

```
[ ] from torch.utils.data import Dataset

    class CustomDataset(Dataset):
      def __init__(self, seqs, visits, gender, race, age, visit_dates):
        self.x = seqs
        self.visit = visits
        self.gender = gender
        self.race = race
        self.age = age
        self.visit_dates = visit_dates

      def __len__(self):
        # your code here
        return len(self.x)

      def __getitem__(self, index):
        # Extract the sequence
        sequence = self.x[index]
        visits = self.visit[index]
        gender = self.gender[index]
        race = self.race[index]
        age = self.age[index]
        visit_dates = self.visit_dates[index]
        # Return the pair (sequence, hf)
        return (sequence, visits, gender, race, age, visit_dates)

    dataset = CustomDataset(seqs, visits, gender, race, age, visit_dates)
```

```
[ ] print(dataset.__getitem__(0))

    #Output
    #([[0, 1, 2, 3, 4, 5, 6, 7], [8, 9, 10, 11, 12, 13, 14, 15], [24, 25, 26, 27, 28, 29, 30, 14, 31, 32, 4, 33, 34], [16, 17, 18, 19, 20, 20, 4, 21, 22, 23]], [1, 2, 3,
```

Data Sampler & Split Data Into Train and Validation Set

We have also created a data sampler to quickly sample the data to test the model training, shapes and evaluation steps.

```
[ ] #Run on sample
    from torch.utils.data import Dataset, SubsetRandomSampler

    seed = 42
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)

    # Define the size of the subset (20% of the dataset)
    #sample_size = 0.2 #Sampling only 20 % of the dataset for model tranining and validation
    sample_size = 1.0 #Sampling all of the dataset 100% for model tranining and validation

    subset_size = int(sample_size * len(dataset))

    # Create a random sampler to sample indices from the dataset
    indices = list(range(len(dataset)))
    np.random.shuffle(indices)  # Shuffle the indices randomly
    subset_indices = indices[:subset_size]  # Take the first subset_size indices

    # Create a SubsetRandomSampler using the subset indices
    subset_sampler = SubsetRandomSampler(subset_indices)
```

```
from torch.utils.data.dataset import random_split

#use subset data and split in 80/20 for train and vel
# Split the subset indices into training and validation indices (80/20 split)
split_index = int(0.8 * len(subset_indices))
train_indices = subset_indices[:split_index]
val_indices = subset_indices[split_index:]

# Create SubsetRandomSamplers for training and validation sets
train_sampler = SubsetRandomSampler(train_indices)
val_sampler = SubsetRandomSampler(val_indices)

print("Length of train dataset:", len(train_sampler))
print("Length of val dataset:", len(val_sampler))

#Length of train dataset: 18564
#Length of val dataset: 4642
```

```
Length of train dataset: 18564
Length of val dataset: 4642
```

Split Data Into Train and Validation Set

Another utility to split the dataset into training and validation sets without sampling.

```
[ ] from torch.utils.data.dataset import random_split

    split = int(len(dataset)*0.8)

    lengths = [split, len(dataset) - split]
    train_dataset, val_dataset = random_split(dataset, lengths)

    print("Length of train dataset:", len(train_dataset))
    print("Length of val dataset:", len(val_dataset))

    #Length of train dataset: 18564
    #Length of val dataset: 4642
```

Data Loader & collate_fn Implementation

Within collate_fu we are computing positional encoding to embed the time, we applied sinusoidal position embedding [2] to the numerical format of visit date (date-specific)

Sample Data Loader and Collate Function

```
[ ] from torch.utils.data import DataLoader
    import math
    import torch.nn.utils.rnn as rnn_utils

    def load_sample_data(dataset, sampler, batch_size, shuffle):
        def collate_fn(data):
            def get_position_encoding(position, d_model):
                """Calculates sinusoidal position encoding for a given position and embedding dimension."""
                pe = torch.zeros(d_model)
                div_term = torch.exp(torch.arange(0, d_model, 2) * (-math.log(10000.0) / d_model))
                #print("position", position)
                #print("div_term",div_term)
                position *= 2 * math.pi
                pe[0::2] = torch.sin(position * div_term)
                pe[1::2] = torch.cos(position * div_term)
                return pe.unsqueeze(0)

            sequences, visits_ids, gender, race, age, visit_dates = zip(*data)
            # Convert gender and race to tensors (optional)
            if gender is not None:
                gender = torch.tensor(gender, dtype=torch.long)
            if race is not None:
                race = torch.tensor(race, dtype=torch.long)
            if age is not None:
                age = torch.tensor(age, dtype=torch.long)

            sequences = [patient[-4:] for patient in sequences]
            visit_dates = [visit_date[-4:] for visit_date in visit_dates]
            visits_ids = [visit_id[:4] for visit_id in visits_ids]

            #positional encoding dim
            d_model = 2

            num_patients = len(sequences)
            num_visits = [len(patient) for patient in sequences]
            num_codes = [len(visit) for patient in sequences for visit in patient]
            max_num_visits = max(num_visits)
            #max_num_visits = 4
            max_num_codes = 5
            pad_value = 0

            visit_numbers = rnn_utils.pad_sequence([torch.tensor(visit) for visit in visits_ids], batch_first=True,padding_value=0)
            num_heads = 1
            x = torch.zeros((num_patients, max_num_visits, max_num_codes), dtype=torch.long)
            masks = torch.zeros((num_patients, num_heads, max_num_visits, max_num_codes), dtype=torch.bool)
            attn_masks = torch.zeros((num_patients, max_num_visits, max_num_visits), dtype=torch.bool)
            position_encodings = torch.zeros((num_patients, max_num_visits, d_model),dtype=torch.float)  # For position encoding

            for i_patient, (patient, visit_date) in enumerate(zip(sequences, visit_dates)):
                valid_visits = [visit for visit in patient if len(visit) > 4]
                if len(valid_visits) >= max_num_visits:
                    for h in range(num_heads):
                      for j_visit, visit in enumerate(valid_visits[:max_num_visits]):

                          last_5_icd_codes = visit[-5:]
```

```
                        x[i_patient, j_visit, :] = torch.tensor(last_5_icd_codes, dtype=torch.long)

                        # Calculate the attention mask
                        attn_mask_row = [1] * (j_visit + 1) + [0] * (max_num_visits - j_visit - 1)
                        attn_masks[i_patient, j_visit] = torch.tensor(attn_mask_row, dtype=torch.bool)

                        # Create mask for the visit (mask all ICD codes in the visit)
                        masks[i_patient, h, j_visit, :len(last_5_icd_codes)] = True

                        if j_visit == len(valid_visits)-1:  # Check if it's the last visit for the patient
                            masks[i_patient, h, j_visit, :] = False

                        # Calculate position encoding based on visit date (assuming YYYY-MM-DD format)
                        year, month, day = map(int, visit_date[j_visit].split('-'))
                        # You can customize the date processing logic based on your data format
                        date_as_float = year + (month - 1) / 12 + day / (365.25 * 12)  # Approximate date as float
                        position_encodings[i_patient, j_visit, :] = get_position_encoding(date_as_float, d_model)

            return (x, attn_masks, masks, visit_numbers, gender, race, age, position_encodings)

        return torch.utils.data.DataLoader(dataset, batch_size=batch_size, collate_fn=collate_fn, sampler=sampler)
```

```
[ ] train_loader = load_sample_data(dataset, train_sampler, batch_size=32, shuffle=True)
    val_loader = load_sample_data(dataset, val_sampler, batch_size=32, shuffle=False)
```

All Data Loader & Collate Function

```
[ ] from torch.utils.data import DataLoader
    import math
    import torch.nn.utils.rnn as rnn_utils

    def load_data(dataset, batch_size, shuffle):
        def collate_fn(data):
            def get_position_encoding(position, d_model):
                """Calculates sinusoidal position encoding for a given position and embedding dimension."""
                pe = torch.zeros(d_model)
                div_term = torch.exp(torch.arange(0, d_model, 2) * (-math.log(10000.0) / d_model))
                #print("position", position)
                #print("div_term",div_term)
                position *= 2 * math.pi
                pe[0::2] = torch.sin(position * div_term)
                pe[1::2] = torch.cos(position * div_term)
                return pe.unsqueeze(0)

            sequences, visits_ids, gender, race, age, visit_dates = zip(*data)
            # Convert gender and race to tensors (optional)
            if gender is not None:
                gender = torch.tensor(gender, dtype=torch.long)
            if race is not None:
                race = torch.tensor(race, dtype=torch.long)
            if age is not None:
                age = torch.tensor(age, dtype=torch.long)

            sequences = [patient[-4:] for patient in sequences]
            visit_dates = [visit_date[-4:] for visit_date in visit_dates]
            visits_ids = [visit_id[:4] for visit_id in visits_ids]

            #positional encoding dim
            d_model = 2
            num_patients = len(sequences)
            num_visits = [len(patient) for patient in sequences]
            num_codes = [len(visit) for patient in sequences for visit in patient]
            max_num_visits = max(num_visits)
            #max_num_visits = 4
            max_num_codes = 5
            pad_value = 0

            visit_numbers = rnn_utils.pad_sequence([torch.tensor(visit) for visit in visits_ids], batch_first=True,padding_value=0)
            num_heads = 1
            x = torch.zeros((num_patients, max_num_visits, max_num_codes), dtype=torch.long)
            masks = torch.zeros((num_patients, num_heads, max_num_visits, max_num_codes), dtype=torch.bool)
            attn_masks = torch.zeros((num_patients, max_num_visits, max_num_visits), dtype=torch.bool)
            position_encodings = torch.zeros((num_patients, max_num_visits, d_model),dtype=torch.float)  # For position encoding

            for i_patient, (patient, visit_date) in enumerate(zip(sequences, visit_dates)):
                valid_visits = [visit for visit in patient if len(visit) > 4]
                #print(valid_visits)
                if len(valid_visits) >= max_num_visits:
                    for h in range(num_heads):
                      for j_visit, visit in enumerate(valid_visits[:max_num_visits]):
                        last_5_icd_codes = visit[-5:]

                        x[i_patient, j_visit, :] = torch.tensor(last_5_icd_codes, dtype=torch.long)

                        attn_mask_row = [1] * (j_visit + 1) + [0] * (max_num_visits - j_visit - 1)
                        attn_masks[i_patient, j_visit] = torch.tensor(attn_mask_row, dtype=torch.bool)

                        masks[i_patient, h, j_visit, :len(last_5_icd_codes)] = True

                        if j_visit == len(valid_visits)-1:  # Check if it's the last visit for the patient
                            masks[i_patient, h, j_visit, :] = False

                        # Calculate position encoding based on visit date (assuming YYYY-MM-DD format)
                        year, month, day = map(int, visit_date[j_visit].split('-'))
                        date_as_float = year + (month - 1) / 12 + day / (365.25 * 12)  # Approximate date as float
                        position_encodings[i_patient, j_visit, :] = get_position_encoding(date_as_float, d_model)

            return (x, attn_masks, masks, visit_numbers, gender, race, age, position_encodings)

        return torch.utils.data.DataLoader(dataset, batch_size=batch_size, collate_fn=collate_fn)
```

```
[ ] train_loader = load_data(train_dataset, batch_size = 32)
```

```
[ ] train_loader = load_data(train_dataset, batch_size = 32)
    val_loader = load_data(val_dataset,  batch_size = 32)
```

```
[ ] #Check the loader and collate function implementation
    loader_iter = iter(train_loader)
    x, attn_masks, masks, visit_numbers, gender, race, age, position_encodings = next(loader_iter)
    print(x, attn_masks , masks,  visit_numbers, gender, race, age, position_encodings)
```

```
[ ] #Check shapes for all the features
    print("x", x.shape)
    print("masks", masks.shape)
    print("visits", visit_numbers.shape)
    print("gender", gender.shape)
    print("race", race.shape)
    print("age", age.shape)
    print("position_encodings", position_encodings.shape)
```

Model

Citation

Yang, Z., Mitra, A., Liu, W. et al. TransformEHR: transformer-based encoder-decoder generative model to enhance prediction of disease outcomes using electronic health records. Nat Commun 14, 7857 (2023). https://doi.org/10.1038/s41467-023-43715-z

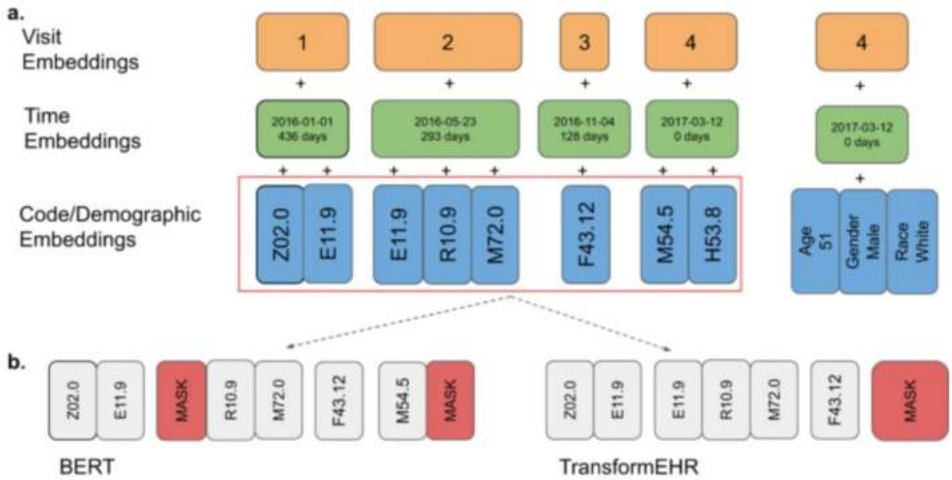
Link to the original paper's repo

- TransformEHR model pre-training codebase is not available.
- Finetuning Code Repo - https://github.com/whaleloops/TransformEHR

Since pre-tranining codebase was not available, it took lot of research for the implementation. Hence we have implemented only the pre-training part of the paper.

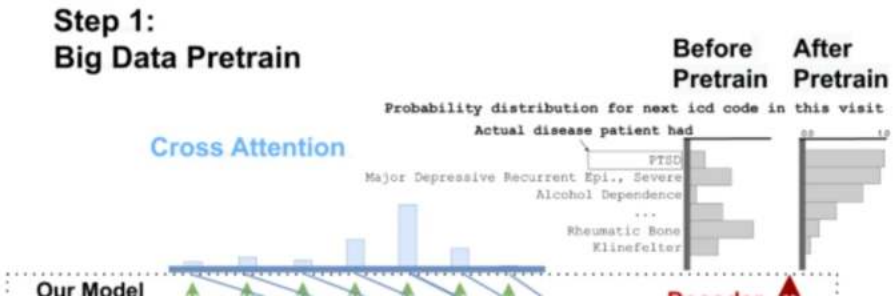
TransformEHR Model Architecture

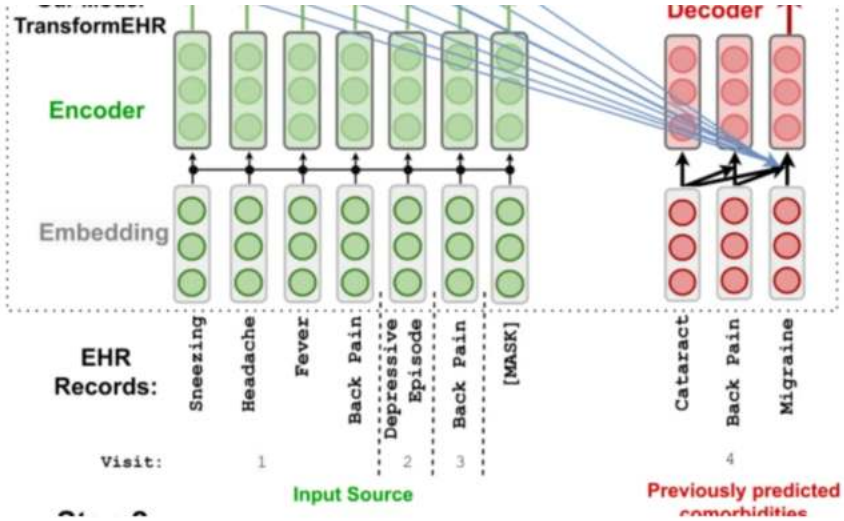
TransformEHR uses a encoder-decoder architecture. The encoder takes in visit, time, and code/demographic embeddings and generates a set of hidden representations for each predictor. TransformEHR then calculates cross-attention over the encoder's created hidden representation. From there, these weighted representations are sent into the decoder, which then creates the ICD codes of the future visit. The decoder generates ICD codes in sequential order of code priority. so for example, we see a primary diagnosis and secondary diagnosis based on primary diagnosis. This process is continued until all diagnoses of a future visit are completed. This process is shown in the picture below.



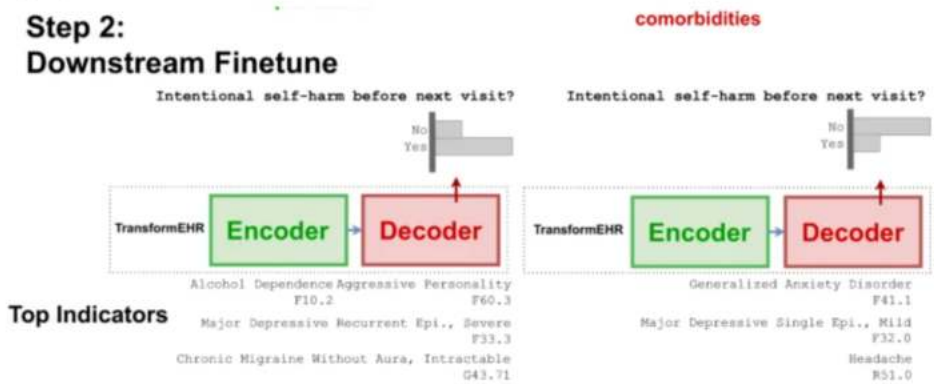
a illustrates the preprocessing of input data from 4 patient visits, with visit embeddings capturing chronological information, time embeddings capturing the specific date, and ICD codes along with demographic data encoded as embeddings. **b** displays various medical history masking schemes: BERT-style masking random codes and TransformEHR-style masking the all codes in a visit.

Pretraining Step





Finetuning Step



TransformEHR Model

This implementation of the TranformEHR model is designed for processing electronic health record (EHR) data. Here's a summary of the key components and functionalities:

Embedding Layers:

- Embedding layers are used for categorical features such as gender and race.
- Continuous features like age and position encodings are also embedded using linear layers.
- Visit embeddings are obtained using an embedding layer based on the number of diagnosis codes.

Concatenation of Embeddings:

- All embeddings (gender, race, age, position encodings, and visit embeddings) are concatenated along the feature dimension.
- The concatenated embeddings are projected to a lower-dimensional space using a linear layer (embedding_projection).

Transformer Encoder:

- Utilizes a transformer encoder with specified parameters like the number of encoder layers (num_encoder_layers) and the number of attention heads (nhead).
- The encoder processes the concatenated embeddings.

Transformer Decoder:

- Employs a transformer decoder with parameters such as the number of decoder layers (num_decoder_layers) and attention heads (nhead).
- Takes the encoder output and the concatenated embeddings as inputs, with masking applied as needed.

Linear Layer for Output:

*A linear layer (linear) is used to project the decoder output to predict probabilities for ICD codes.

Forward Method:

- The forward method takes input data (x), masks for padding (masks), as well as gender, race, age, and position encodings.
- It performs the embedding, concatenation, projection, transformer encoding, decoding, and output projection steps.

Model Initialization:

- The model is initialized with specified parameters such as the number of gender classes, race classes, and the maximum number of visits and diagnosis codes.

Overall, this implementation encapsulates the key components of the TranformEHR model for processing EHR data with transformer-based encoder-decoder architecture and cross attentions.

Model v1 - TransformEHR_M1

For better understanding of the model architecture, masking strategy and debugging, first we experimented with pre-training the model on

For better understanding of the model architecture, masking strategy and debugging, first we experimented with pre-training the model on sequence of ICD-Codes only

Features

- sequence of icd_diagnosis codes.

Model Parameters

- Batch Size = 32
- Learning Rate = 0.0001
- Number of Head = 1
- Encoder Layer = 1
- Decoder Layer = 1
- Epoc = 10
- Batch First = True
- Norm First = True

Model Implementation - TranformEHR_M1

```
[ ] # Define the number of classes for each categorical feature
    num_gender_classes = 2
    num_race_classes = 33
    # Define the maximum number of visits and diagnosis codes
    #num_visits = [len(visit) for visit in visits]
    #max_num_visits = max(num_visits)

    max_num_visits = 4
    max_num_codes = 5

    def get_encoder_mask(batch_size, seq_length):
        # Create a square matrix with ones in the lower triangle (including the diagonal)
        mask = torch.tril(torch.ones(seq_length, seq_length))
        # Expand to match the batch size
        mask = mask.unsqueeze(0).unsqueeze(1).expand(batch_size, 1, seq_length, seq_length)
        return mask

    class TranformEHR_M1(nn.Module):
        def __init__(self, num_gender_classes, num_race_classes, num_visits, num_code, nhead, num_encoder_layers,
                     num_decoder_layers, embedding_dim=128):
            super().__init__()
            self.embedding_dim = embedding_dim
            self.concatenated_dim = embedding_dim * 1
            self.projected_dim = embedding_dim
            self.num_heads = nhead

            # Define the embedding layers
            self.visit_number_embedding = nn.Embedding(num_embeddings=num_visits, embedding_dim=embedding_dim)
            self.gender_embedding = nn.Embedding(num_gender_classes, embedding_dim)
            self.race_embedding = nn.Embedding(num_race_classes, embedding_dim)

            # Define the embeddings for other continuous features (age, position_encodings)
            self.age_embedding = nn.Linear(1, embedding_dim)  #age is a continuous feature
            self.position_encodings_embedding = nn.Linear(2, embedding_dim)  # position_encodings has 2 dimensions

            self.visit_embedding = nn.Embedding(num_embeddings=num_code, embedding_dim=embedding_dim)

            self.embedding_projection = nn.Linear(self.concatenated_dim, self.projected_dim)

            # Transformer encoder
            encoder_layer = nn.TransformerEncoderLayer(d_model=embedding_dim, nhead=nhead, batch_first=True, norm_first=True)

            self.transformer_encoder = nn.TransformerEncoder(encoder_layer, num_layers=num_encoder_layers)

            # Transformer decoder
            decoder_layer = nn.TransformerDecoderLayer(d_model=self.projected_dim, nhead=nhead, batch_first=True, norm_first=True)

            self.transformer_decoder = nn.TransformerDecoder(decoder_layer, num_layers=num_decoder_layers)
            # Linear layer to project decoder output to ICD code probabilities
            self.linear = nn.Linear(self.projected_dim, num_code)


        def forward(self, x, attn_masks, masks, visit_numbers, gender, race, age, position_encodings):

            max_visit_number = self.visit_number_embedding.num_embeddings - 1  # Get the max allowed index
            visit_numbers = torch.clamp(visit_numbers, 0, max_visit_number)  # Clamp values to valid range
            embedded_visits_number = self.visit_number_embedding(visit_numbers)
            embedded_gender = self.gender_embedding(gender)
            embedded_race = self.race_embedding(race)
            embedded_age = self.age_embedding(age.float().unsqueeze(-1))
            embedded_positional_encodings = self.position_encodings_embedding(position_encodings.float())

            embedded_x = self.visit_embedding(x)

            """embedded_positional_encodings = embedded_positional_encodings.unsqueeze(2).expand(-1, -1, embedded_x.size(2),-1)
            embedded_age = embedded_age.unsqueeze(1).unsqueeze(2).expand(-1, embedded_x.size(1), embedded_x.size(2), -1)
            embedded_race = embedded_race.unsqueeze(1).unsqueeze(2).expand(-1, embedded_x.size(1), embedded_x.size(2), -1)
            embedded_gender = embedded_gender.unsqueeze(1).unsqueeze(2).expand(-1, embedded_x.size(1), embedded_x.size(2),-1)
            embedded_visits_number = embedded_visits_number.unsqueeze(2).expand(-1, embedded_x.size(1), embedded_x.size(2),-1)"""

            embedded_input = embedded_x.reshape(embedded_x.size(0), -1, self.projected_dim)

            #Compute the attn_mask
            batch_size = x.size(0)
            new_masks = get_encoder_mask(batch_size, 20)
            new_masks = new_masks.reshape(batch_size*self.num_heads,20, 20)

            #Apply Encoder
            encoder_output = self.transformer_encoder(embedded_input, mask = new_masks)
```

```
        # Apply transformer decoder
        print("embedded_input.shape - ",embedded_input.shape)
        print("encoder_output.shape - ",encoder_output.shape)
        decoder_output = self.transformer_decoder(embedded_input, encoder_output, tgt_mask=new_masks)

        # Calculate logits
        logits = self.linear(decoder_output)

        return logits

# Instantiate the model_m1
model_m1 = TranformEHR_M1(num_gender_classes, num_race_classes, num_visits=max_num_visits, num_code=len(icd_codes_types),nhead=1, num_encoder_layers=1, num_decoder_l
```

Model v2 - TransformEHR_M2

In Model V2 - we have added more features -

Features

- Visit embeddings +
- Time (Visit Date) embeddings +
- Demographic embeddings (Age, Gender and Race) +
- ICD-Code embeddings

Model Parameters

- Batch Size = 32
- Learning Rate = 0.0001
- Number of Head = 1
- Encoder Layer = 1
- Decoder Layer = 1
- Epoc = 10
- Batch First = True
- Norm First = True

Model Implementation - TranformEHR_M2

```
[ ] # Define the number of classes for each categorical feature
    num_gender_classes = 2
    num_race_classes = 33
    # Define the maximum number of visits and diagnosis codes
    #num_visits = [len(visit) for visit in visits]
    #max_num_visits = max(num_visits)
    max_num_visits = 4
    max_num_codes = 5

    def get_encoder_mask(batch_size, seq_length):
        # Create a square matrix with ones in the lower triangle (including the diagonal)
        mask = torch.tril(torch.ones(seq_length, seq_length))
        # Expand to match the batch size
        mask = mask.unsqueeze(0).unsqueeze(1).expand(batch_size, 1, seq_length, seq_length)
        return mask

    class TranformEHR_M2(nn.Module):
        def __init__(self, num_gender_classes, num_race_classes, num_visits, num_code, nhead, num_encoder_layers,
                     num_decoder_layers, embedding_dim=128):
            super().__init__()
            self.embedding_dim = embedding_dim
            self.concatenated_dim = embedding_dim * 6
            self.projected_dim = embedding_dim
            self.num_heads = nhead

            # Define the embedding layers
            self.visit_number_embedding = nn.Embedding(num_embeddings=num_visits, embedding_dim=embedding_dim)
            self.gender_embedding = nn.Embedding(num_gender_classes, embedding_dim)
            self.race_embedding = nn.Embedding(num_race_classes, embedding_dim)

            # Define the embeddings for other continuous features (age, position_encodings)
            self.age_embedding = nn.Linear(1, embedding_dim)  # Assuming age is a continuous feature
            self.position_encodings_embedding = nn.Linear(2, embedding_dim)  # Assuming position_encodings has 2 dimensions
            self.visit_embedding = nn.Embedding(num_embeddings=num_code, embedding_dim=embedding_dim)

            self.embedding_projection = nn.Linear(self.concatenated_dim, self.projected_dim)

            # Transformer encoder
            encoder_layer = nn.TransformerEncoderLayer(d_model=embedding_dim, nhead=nhead, batch_first=True, norm_first=True)

            self.transformer_encoder = nn.TransformerEncoder(encoder_layer, num_layers=num_encoder_layers)

            # Transformer decoder
            decoder_layer = nn.TransformerDecoderLayer(d_model=self.projected_dim, nhead=nhead, batch_first=True, norm_first=True)

            self.transformer_decoder = nn.TransformerDecoder(decoder_layer, num_layers=num_decoder_layers)

            # Linear layer to project decoder output to ICD code probabilities
            self.linear = nn.Linear(self.projected_dim, num_code)

        def forward(self, x, attn_masks, masks, visit_numbers, gender, race, age, position_encodings):

            max_visit_number = self.visit_number_embedding.num_embeddings - 1  # Get the max allowed index
            visit_numbers = torch.clamp(visit_numbers, 0, max_visit_number)  # Clamp values to valid range
            embedded_visits_number = self.visit_number_embedding(visit_numbers)
            embedded_gender = self.gender_embedding(gender)
            embedded_race = self.race_embedding(race)
            embedded_age = self.age_embedding(age.float().unsqueeze(-1))
            embedded_positional_encodings = self.position_encodings_embedding(position_encodings.float())
            embedded_x = self.visit_embedding(x)
```

```python
        # Concatenate all embeddings
        embedded_positional_encodings = embedded_positional_encodings.unsqueeze(2).expand(-1, -1, embedded_x.size(2), -1)
        embedded_age = embedded_age.unsqueeze(1).unsqueeze(2).expand(-1, embedded_x.size(1), embedded_x.size(2), -1)
        embedded_race = embedded_race.unsqueeze(1).unsqueeze(2).expand(-1, embedded_x.size(1), embedded_x.size(2), -1)
        embedded_gender = embedded_gender.unsqueeze(1).unsqueeze(2).expand(-1, embedded_x.size(1), embedded_x.size(2),-1)
        embedded_visits_number = embedded_visits_number.unsqueeze(2).expand(-1, embedded_x.size(1), embedded_x.size(2),-1)

        embedded_input = torch.cat((embedded_x, embedded_visits_number, embedded_positional_encodings, embedded_age,embedded_race, embedded_gender), dim=-1)

        embedded_input = embedded_x.reshape(embedded_input.size(0), -1, self.projected_dim)

        #Calculate attn_mask
        batch_size = x.size(0)
        new_masks = get_encoder_mask(batch_size, 20)
        new_masks = new_masks.reshape(batch_size*self.num_heads,20, 20)

        #Apply Transformer Encoder
        encoder_output = self.transformer_encoder(embedded_input, mask = new_masks)

        # Apply Transformer Decoder
        print("embedded_input.shape - ",embedded_input.shape)
        print("encoder_output.shape - ",encoder_output.shape)
        decoder_output = self.transformer_decoder(embedded_input, encoder_output, tgt_mask=new_masks)

        logits = self.linear(decoder_output)

        return logits

# Instantiate the model_m2
model_m2 = TranformEHR_M2(num_gender_classes, num_race_classes, num_visits=max_num_visits, num_code=len(icd_codes_types),nhead=1, num_encoder_layers=1, num_decoder_l
```

Model Training & Eveluation

This code defines functions to train and evaluate a model using PyTorch for a task involving the TransformEHR architecture. Here's a breakdown of each part:

Loss Function and Optimizer:

- criterion = nn.CrossEntropyLoss(): Defines the cross-entropy loss function, commonly used for classification tasks.
- optimizer = torch.optim.Adam(model.parameters()): Initializes the Adam optimizer to update the model parameters during training.

Training Function (train):

- Takes input model (the TransformEHR model), train_data_loader (dataloader for training data), and epochs (number of training epochs).
- Sets the model to training mode (model.train()).
- Iterates through each epoch and batch of data, computes the loss using the defined loss function, performs backpropagation, and updates the model parameters.
- Optionally prints training progress.

Evaluation Function (eval):

- Takes input model (the TransformEHR model) and val_data_loader (dataloader for validation data).
- Sets the model to evaluation mode (model.eval()).
- Disables gradient calculation (torch.no_grad()) for efficiency during evaluation.
- Computes the average loss on the validation data by iterating through batches and calculating the loss using the same criterion as in training.

Example Usage:

- Calls the train function to train the model for 10 epochs using the training data (train_loader).
- Calls the eval function to evaluate the trained model using validation data (val_loader) and prints the average evaluation loss.

Overall, this code snippet provides a structured way to train and evaluate a model using PyTorch, suitable for tasks like the TransformEHR architecture where data is fed in batches through dataloaders, and the model's performance is assessed using a loss function.

Hyperparams

- Batch Size = 32
- Learning Rate = 0.0001
- Hidden Size = 128

Other Parameters

- Number of Head = 1
- Encoder Layer = 1
- Decoder Layer = 1
- Epochs = 10
- Batch First = True
- Norm First = True

Computational requirements

- Hardware -
  - Colab Enterprise Vertex AI Runtime
  - Machine type - e2-highmem-16
  - CPU - 16
  - Memory - 128
- Average Runtime for each epoch = ~3 minute
- Total Execution Time - ~27 minutes
- GPU hrs used = Model was trained using CPU's
- Training epochs = 10

Training & Evaluation - Model M1

```
[ ] import torch
    from torch import nn
    from torch.utils.data import DataLoader
    from sklearn.metrics import precision_score, roc_auc_score, average_precision_score, f1_score, accuracy_score, precision_recall_fscore_support

    #for mac - use mps
    #for all other - use cuda

    #device = torch.device("cuda" if torch.backends.mps.is_available() else "cpu")
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    model_m1.to(device)

    # Define your loss function (e.g., cross-entropy)
    criterion = nn.CrossEntropyLoss()

    # Define your optimizer (e.g., Adam)
    optimizer = torch.optim.Adam(model_m1.parameters(), lr=0.0001)
    #optimizer = torch.optim.Adam(model.parameters(), lr=1e-3, weight_decay=1e-5)
    #optimizer = torch.optim.Adam(model.parameters(),lr=1e-4)

    log_interval = 5

    train_losses = []  # List of training losses over epochs
    eval_losses = []  # List of evaluation losses over epochs
    accuracies = []  # List of accuracies over epochs
    precisions = []  # List of precisions over epochs
    recalls = []  # List of recalls over epochs
    f1_scores = []  # List of F1-scores over epochs
    topk_accuracies = []  # List of top-k accuracies over epochs
    precisions_macro = []  # List of macro precisions over epochs
    precisions_weighted = []  # List of weighted precisions over epochs
    f1_scores_macro = []  # List of macro F1-scores over epochs
    f1_scores_weighted = []  # List of weighted F1-scores over epochs
    roc_auc_scores_R = []  # List of ROC AUC scores over epochs
    roc_auc_scores_O = []
    average_precision_scores = []  # List of average precision scores over epochs
    all_probabilities_scores = []


    def train_eval(model, train_data_loader, val_data_loader, epochs):

        for epoch in range(epochs):
            print(f"Epoch {epoch + 1} of {epochs}")
            model.train() # Set model to training mode
            epoch_loss = 0.0
            for batch_idx, batch in enumerate(train_loader):
                print(batch_idx)
                x, attn_masks, masks, visit_numbers, gender, race, age, position_encodings = [item.to(device) for item in batch]

                # Forward pass
                logits = model(x)

                loss = criterion(logits.view(-1, logits.size(-1)), x.view(-1))

                # Backward pass and update parameters
                optimizer.zero_grad()
                loss.backward()
                optimizer.step()
                epoch_loss += loss.item()
            # Calculate average epoch loss
            epoch_loss /= len(train_data_loader)
            print("training_epoc_loss - ", epoch_loss)
            train_losses.append(epoch_loss)

            print("starting evaluation")
            model.eval()
            with torch.no_grad():  # Disable gradient calculation for efficiency
                eval_epoch_loss = 0.0
                all_predictions = []
                all_probabilities = []
                all_targets = []
                for x, attn_masks, masks, visit_numbers, gender, race, age, position_encodings in val_data_loader:
                    #x_batch_size = x.size(0)
                    #if not x_batch_size < 32:
                    x, attn_masks, masks, visit_numbers, gender, race, age, position_encodings = x.to(device), attn_masks.to(device), masks.to(device), \
                    visit_numbers.to(device), gender.to(device), race.to(device), age.to(device), \
                    position_encodings.to(device)

                    # Forward pass
                    print(x.shape)
                    logits = model(x)
                    loss = criterion(logits.view(-1, logits.size(-1)), x.view(-1))
                    eval_epoch_loss += loss.item()

                    predicted_codes = torch.argmax(logits, dim=-1)
                    all_predictions.append(predicted_codes.cpu().numpy().flatten())
                    all_targets.append(x.cpu().numpy().flatten())

                    # Convert logits to probabilities
                    """predicted_probs = torch.softmax(logits, dim=-1)
                    all_probabilities_scores.append(predicted_probs)
                    print("predicted_probs shape", predicted_probs.shape)
                    all_probabilities.append(predicted_probs.cpu().numpy())"""

                # Calculate average epoch loss
                eval_epoch_loss /= len(val_data_loader)
                eval_losses.append(eval_epoch_loss)

                # Calculate accuracy
                all_predictions = np.concatenate(all_predictions)
                """all_probabilities_np = np.concatenate(all_probabilities, axis=0)
```

```
            # Average the predicted probabilities across the sequence length dimension
            average_probabilities = np.mean(all_probabilities_np, axis=1)
            # Flatten the probabilities
            flattened_probabilities = average_probabilities.reshape(-1)"""
            all_targets = np.concatenate(all_targets)
            accuracy = accuracy_score(all_targets, all_predictions)
            accuracies.append(accuracy)

            # Calculate precision, recall, and F1-score
            precision, recall, f1, _ = precision_recall_fscore_support(all_targets, all_predictions, average='macro')
            precisions.append(precision)
            recalls.append(recall)
            f1_scores.append(f1)
            # Compute evaluation metrics
            precision_macro = precision_score(all_targets, all_predictions.round(), average='macro')
            precision_weighted = precision_score(all_targets, all_predictions.round(), average='weighted')
            precisions_macro.append(precision_macro)  # List of macro precisions over epochs
            precisions_weighted.append(precision_weighted)  # List of weighted precisions over epochs

            f1_score_macro = f1_score(all_targets, all_predictions.round(), average='macro')
            f1_score_weighted = f1_score(all_targets, all_predictions.round(), average='weighted')
            f1_scores_macro.append(f1_score_macro)  # List of macro F1-scores over epochs
            f1_scores_weighted.append(f1_score_weighted)  # List of weighted F1-scores over epochs

            #roc_auc_R = roc_auc_score(all_targets, flattened_probabilities, multi_class='ovr')
            #roc_auc_O = roc_auc_score(all_targets, flattened_probabilities, multi_class='ovo')
            #roc_auc_scores_R.append(roc_auc_R)  # List of ROC AUC scores over epochs
            #roc_auc_scores_O.append(roc_auc_O)  # List of ROC AUC scores over epochs

            #avg_precision = average_precision_score(all_targets, all_predictions) #default is - average='macro
            #average_precision_scores.append(avg_precision)  # List of average precision scores over epochs
        print(f"Epoch [{epoch + 1}/{epochs}], Average Training Loss: {epoch_loss:.4f},"
              f"Average Evaluation Loss : {eval_epoch_loss:.4f}, Accuracy: {accuracy:.4f},"
              f"Precision: {precision:.4f}, Recall: {recall:.4f}, F1-score: {f1:.4f}, "
              f"Precision Macro: {precision_macro:.4f}, Precision Weighted: {precision_weighted:.4f}, "
              f"F1-score Macro: {f1_score_macro:.4f}, F1-score Weighted: {f1_score_weighted:.4f}")
```

```
[ ] #Train model_m1
    import time
    start = time.time()
    print(start)
    train_eval(model_m1, train_loader, val_loader, epochs=10)
    end = time.time()
    print(end)
    print(end - start)
```

Plot evaluation metrics for Model M1

```
[ ] #Plot
    import matplotlib.pyplot as plt

    def plot_metrics(headline, train_losses, eval_losses, accuracies, precisions,
                     recalls, f1_scores, precisions_macro, precisions_weighted,
                     f1_scores_macro, f1_scores_weighted, roc_auc_scores_O,
                     roc_auc_scores_R, average_precision_scores):
        # Plot losses
        epochs = range(1, len(train_losses) + 1)
        plt.suptitle(headline, fontsize=16)
        # Plot losses
        plt.figure(figsize=(10, 3))

        plt.subplot(1, 3, 1)
        plt.plot(epochs, train_losses, label='Training Loss')
        plt.xlabel('Epochs')
        plt.ylabel('Loss')
        plt.title('Training Loss')
        plt.legend()

        plt.subplot(1, 3, 2)
        plt.plot(epochs, eval_losses, label='Evaluation Loss', color='red')
        plt.xlabel('Epochs')
        plt.ylabel('Loss')
        plt.title('Evaluation Loss')
        plt.legend()

        # Plot evaluation metrics
        plt.subplot(1, 3, 3)
        plt.plot(epochs, accuracies, label='Accuracy')
        plt.plot(epochs, precisions, label='Precision')
        plt.plot(epochs, recalls, label='Recall')
        plt.plot(epochs, f1_scores, label='F1-score')
        #plt.plot(epochs, topk_accuracies, label='Top-k Accuracy')
        plt.xlabel('Epochs')
        plt.ylabel('Metrics')
        plt.title('Evaluation Metrics')
        plt.legend()

        plt.figure(figsize=(10, 3))
        plt.subplot(1, 3, 1)
        plt.plot(epochs, precisions_macro, label='Precisions Macro')
        plt.xlabel('Epochs')
        plt.ylabel('Precisions')
        plt.title('Precisions [ Macro]')
        plt.legend()

        plt.subplot(1, 3, 2)
        plt.plot(epochs, precisions_weighted, label='Precisions Weighted')
        plt.xlabel('Epochs')
        plt.ylabel('Precisions')
        plt.title('Precisions [ Weighted]')
        plt.legend()
```

```
    plt.subplot(1, 3, 3)
    plt.plot(epochs, f1_scores_macro, label='F1 Macro')
    plt.xlabel('Epochs')
    plt.ylabel('F1-Score')
    plt.title('F1-Score [ Macro]')
    plt.legend()

    plt.figure(figsize=(10, 3))
    plt.subplot(1, 3, 1)
    plt.plot(epochs, f1_scores_weighted, label='F1 Weighted')
    plt.xlabel('Epochs')
    plt.ylabel('F1-Score')
    plt.title('F1-Score [ Weighted]')
    plt.legend()

    """#average_precision_scores
    plt3.subplot(1, 3, 2)
    plt3.plot(epochs, average_precision_scores, label='Average Precision')
    plt3.xlabel('Epochs')
    plt3.ylabel('Average Precision')
    plt3.title('Average Precision')
    plt3.legend()

    plt3.subplot(1, 3, 3)
    plt3.plot(epochs, roc_auc_scores_O, label='ROC-AUC-OVO')
    plt3.xlabel('Epochs')
    plt3.ylabel('ROC-AUC-OVO')
    plt3.title('ROC-AUC [One-vs-one]')
    plt3.legend()

    plt.subplot(3, 3, 2)
    plt.plot(epochs, roc_auc_scores_R, label='ROC-AUC-OVR')
    plt.xlabel('Epochs')
    plt.ylabel('ROC-AUC-OVR')
    plt.title('ROC-AUC [One-vs-rest]')
    plt.legend()"""

    #plt.tight_layout()
    plt.show()
```

```
[ ] headlind_1 = "TransformEHR - Feature -[ICD Codes only], Num Head - 1, Encoder Layer - 1, Decoder Layer - 1, Learning Rate - 0.0001, Batch Size - 32"
    plot_metrics(headlind_1, train_losses, eval_losses, accuracies, precisions, recalls,
                 f1_scores, precisions_macro, precisions_weighted, f1_scores_macro,
                 f1_scores_weighted, roc_auc_scores_O, roc_auc_scores_R, average_precision_scores)
```

Create Check-Point for Model M1 for Finetuning

```
[ ] #Create a checkpoint dictionary:
    checkpoint = {
        'model_state_dict': model_m1.state_dict(),
        'optimizer_state_dict': optimizer.state_dict(),
        'epoch': 10,  # Current training epoch
        'loss': train_losses   # Current training loss (optional)
    }

    filename = 'checkpoint_model1.pth'  # Create a unique filename
    torch.save(checkpoint, filename)
```

Training & Evaluation - Model M2

```
[ ] import torch
    from torch import nn
    from torch.utils.data import DataLoader
    from sklearn.metrics import precision_score, roc_auc_score, average_precision_score, f1_score, accuracy_score, precision_recall_fscore_support

    #for mac - use mps
    #for all other - use cuda

    #device = torch.device("cuda" if torch.backends.mps.is_available() else "cpu")
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    model_m2.to(device)

    # Define your loss function (e.g., cross-entropy)
    criterion = nn.CrossEntropyLoss()

    # Define your optimizer (e.g., Adam)
    optimizer = torch.optim.Adam(model_m2.parameters(), lr=0.0001)
    #optimizer = torch.optim.Adam(model_m2.parameters(), lr=1e-3, weight_decay=1e-5)
    #optimizer = torch.optim.Adam(model_m2.parameters(),lr=1e-4)

    log_interval = 5

    train_losses = []  # List of training losses over epochs
    eval_losses = []  # List of evaluation losses over epochs
    accuracies = []  # List of accuracies over epochs
    precisions = []  # List of precisions over epochs
    recalls = []  # List of recalls over epochs
    f1_scores = []  # List of F1-scores over epochs
    topk_accuracies = []  # List of top-k accuracies over epochs
    precisions_macro = []  # List of macro precisions over epochs
    precisions_weighted = []  # List of weighted precisions over epochs
    f1_scores_macro = []  # List of macro F1-scores over epochs
    f1_scores_weighted = []  # List of weighted F1-scores over epochs
    roc_auc_scores_R = []  # List of ROC AUC scores over epochs
    roc_auc_scores_O = []
    average_precision_scores = []  # List of average precision scores over epochs
    all_probabilities_scores = []


    def train_eval(model, train_data_loader, val_data_loader, epochs):
```

```
    for epoch in range(epochs):
        print(f"Epoch {epoch + 1} of {epochs}")
        model.train() # Set model to training mode
        epoch_loss = 0.0
        for batch_idx, batch in enumerate(train_loader):
            print(batch_idx)
            x, attn_masks, masks, visit_numbers, gender, race, age, position_encodings = [item.to(device) for item in batch]

            # Forward pass
            logits = model(x, attn_masks, masks, visit_numbers, gender, race, age, position_encodings)

            loss = criterion(logits.view(-1, logits.size(-1)), x.view(-1))

            # Backward pass and update parameters
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            epoch_loss += loss.item()
        # Calculate average epoch loss
        epoch_loss /= len(train_data_loader)
        print("training_epoc_loss - ", epoch_loss)
        train_losses.append(epoch_loss)

        print("starting evaluation")
        model.eval()
        with torch.no_grad():  # Disable gradient calculation for efficiency
            eval_epoch_loss = 0.0
            all_predictions = []
            all_probabilities = []
            all_targets = []
            for x, attn_masks, masks, visit_numbers, gender, race, age, position_encodings in val_data_loader:
                #x_batch_size = x.size(0)
                #if not x_batch_size < 32:
                x, attn_masks, masks, visit_numbers, gender, race, age, position_encodings = x.to(device), attn_masks.to(device), masks.to(device), \
                visit_numbers.to(device), gender.to(device), race.to(device), age.to(device), \
                position_encodings.to(device)

                # Forward pass
                print(x.shape)
                logits = model(x, attn_masks, masks, visit_numbers, gender, race, age, position_encodings)
                loss = criterion(logits.view(-1, logits.size(-1)), x.view(-1))
                eval_epoch_loss += loss.item()

                predicted_codes = torch.argmax(logits, dim=-1)
                all_predictions.append(predicted_codes.cpu().numpy().flatten())
                all_targets.append(x.cpu().numpy().flatten())

                # Convert logits to probabilities
                """predicted_probs = torch.softmax(logits, dim=-1)
                all_probabilities_scores.append(predicted_probs)
                print("predicted_probs shape", predicted_probs.shape)
                all_probabilities.append(predicted_probs.cpu().numpy())"""

            # Calculate average epoch loss
            eval_epoch_loss /= len(val_data_loader)
            eval_losses.append(eval_epoch_loss)

            # Calculate accuracy
            all_predictions = np.concatenate(all_predictions)
            """all_probabilities_np = np.concatenate(all_probabilities, axis=0)
            # Average the predicted probabilities across the sequence length dimension
            average_probabilities = np.mean(all_probabilities_np, axis=1)
            # Flatten the probabilities
            flattened_probabilities = average_probabilities.reshape(-1)"""
            all_targets = np.concatenate(all_targets)
            accuracy = accuracy_score(all_targets, all_predictions)
            accuracies.append(accuracy)

            # Calculate precision, recall, and F1-score
            precision, recall, f1, _ = precision_recall_fscore_support(all_targets, all_predictions, average='macro')
            precisions.append(precision)
            recalls.append(recall)
            f1_scores.append(f1)
            # Compute evaluation metrics
            precision_macro = precision_score(all_targets, all_predictions.round(), average='macro')
            precision_weighted = precision_score(all_targets, all_predictions.round(), average='weighted')
            precisions_macro.append(precision_macro)  # List of macro precisions over epochs
            precisions_weighted.append(precision_weighted)  # List of weighted precisions over epochs

            f1_score_macro = f1_score(all_targets, all_predictions.round(), average='macro')
            f1_score_weighted = f1_score(all_targets, all_predictions.round(), average='weighted')
            f1_scores_macro.append(f1_score_macro)  # List of macro F1-scores over epochs
            f1_scores_weighted.append(f1_score_weighted)  # List of weighted F1-scores over epochs

            #roc_auc_R = roc_auc_score(all_targets, flattened_probabilities, multi_class='ovr')
            #roc_auc_O = roc_auc_score(all_targets, flattened_probabilities, multi_class='ovo')
            #roc_auc_scores_R.append(roc_auc_R)  # List of ROC AUC scores over epochs
            #roc_auc_scores_O.append(roc_auc_O)  # List of ROC AUC scores over epochs

            #avg_precision = average_precision_score(all_targets, all_predictions) #default is - average='macro
            #average_precision_scores.append(avg_precision)  # List of average precision scores over epochs
        print(f"Epoch [{epoch + 1}/{epochs}], Average Training Loss: {epoch_loss:.4f},"
              f"Average Evaluation Loss : {eval_epoch_loss:.4f}, Accuracy: {accuracy:.4f},"
              f"Precision: {precision:.4f}, Recall: {recall:.4f}, F1-score: {f1:.4f}, "
              f"Precision Macro: {precision_macro:.4f}, Precision Weighted: {precision_weighted:.4f}, "
              f"F1-score Macro: {f1_score_macro:.4f}, F1-score Weighted: {f1_score_weighted:.4f}")
```

```
[ ] #Train model_m2
    import time
    start = time.time()
    print(start)
    train_eval(model_m2, train_loader, val_loader, epochs=10)
    end = time.time()
    print(end)
```

```
print(end - start)
```

Plot evaluation metrics for Model M2

```
[ ] # Model #2
headlind_2 = "TransformEHR - Feature - all features], Num Head - 1, Encoder Layer - 1, Decoder Layer - 1, Learning Rate - 0.0001, Batch Size - 32"
plot_metrics(headlind_2, train_losses, eval_losses, accuracies, precisions, recalls,
             f1_scores, precisions_macro, precisions_weighted, f1_scores_macro,
             f1_scores_weighted, roc_auc_scores_O, roc_auc_scores_R, average_precision_scores)
```

Create CheckPoints for Model M2 for Finetuning

```
[ ] #Create a checkpoint dictionary:
checkpoint = {
    'model_state_dict': model_m2.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
    'epoch': 10,  # Current training epoch
    'loss': train_losses   # Current training loss (optional)
}

filename = 'checkpoint_model2.pth'  # Create a unique filename
torch.save(checkpoint, filename)
```

Disease/outcome agnostic prediction: AUROC scores on different pretraining objectives for the 10 common and 10 uncommon diseases

```
[ ] common_outcomes["ICD-10-CM Code ID"] = [icd_codes_types[code] for code in common_outcomes["ICD-10-CM Code"]]
print(common_outcomes)
```

```
{'ICD-10-CM Code': ['I10', 'E785', 'Z87891', 'K219', 'F329', 'I2510', 'F419', 'N179', 'Z794', 'Z7901'], 'Description': ['Essential (primary) hypertension', 'Hyperlipi
```

```
[ ] uncommon_outcomes["ICD-10-CM Code ID"] = [icd_codes_types[code] for code in uncommon_outcomes["ICD-10-CM Code"]]
print(uncommon_outcomes)
```

```
{'ICD-10-CM Code': ['N94.6', 'T47.1X5D', 'O30.033', 'I70234', 'I95.2', 'Z34.83', 'C8518', 'L89.891', 'D126', 'I201'], 'Description': ['Dyspareunia, unspecified', 'Poi
```

```
[ ] import torch
from sklearn.metrics import precision_score, roc_auc_score, average_precision_score
import matplotlib.pyplot as plt


def evaluate_model(model, data_loader, outcomes):
  device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
  model.to(device)
  model.eval()
  disease_codes = outcomes['ICD-10-CM Code ID']
  num_diseases = len(disease_codes)
  aurocs = []

  predictions = {code: [] for code in disease_codes}
  targets = {code: [] for code in disease_codes}

  num = 0
  with torch.no_grad():
    for x, attn_masks, masks, visit_numbers, gender, race, age, position_encodings in data_loader:
      x = x.to(device)
      # Run model
      logits = model(x, attn_masks, masks, visit_numbers, gender, race, age, position_encodings)
      probs = torch.sigmoid(logits[:, -1, :])  # Get the last visit probabilities

      # Extract ICD codes for the last visit for each sample in the batch
      last_visit_codes = x[:, -1, :]  # Shape: (batch_size, num_icd_codes_per_visit)

      for i, code_id in enumerate(disease_codes):
        # Check if each sample's last visit ICD codes contain the current disease code
        code_mask = (last_visit_codes == code_id).any(dim=1)  # Shape: (batch_size)
        batch_targets = code_mask.float().cpu().numpy()
        # if (num == 0):
        #   print("code_id:", code_id)
        #   print("last_visit_codes:", last_visit_codes)
        #   print("last_visit_codes shape:", last_visit_codes.shape)
        #   print("probs:", probs)
        #   print("probs shape:", probs.shape)
        #   print("batch_targets:", batch_targets, batch_targets.shape)

        # Collect targets and predictions
        targets[code_id].extend(batch_targets)
        predictions[code_id].extend(probs[:, i].cpu().numpy())
      num += 1

  # Calculate AUROC for each disease
  for code_id in disease_codes:
    if len(np.unique(targets[code_id])) > 1:
      auroc = roc_auc_score(targets[code_id], predictions[code_id])
      aurocs.append((outcomes['Description'][disease_codes.index(code_id)], auroc))

  return aurocs
```

Model M1 - AUROC scores on different pretraining objectives for the 10 common and 10 uncommon diseases

```
[ ] data_loader = load_data(dataset, batch_size = 32, shuffle=False)
```

```
[ ] #Load Model M1

model_m1_checkpoint_path = '/content/drive/MyDrive/DLH/Resources/CheckPoint/checkpoint_may05-full-model1.pth'

# Load model checkpoint
```

```
model_m1_checkpoint = torch.load(model_m1_checkpoint_path)
model_m1_state_dict = model_m1_checkpoint['model_state_dict']
```

```
[ ] model_m1 = TranformEHR_M1(num_gender_classes, num_race_classes, num_visits=max_num_visits, num_code=len(icd_codes_types),nhead=1, num_encoder_layers=1, num_decoder_l

    # Load the model's state dictionary
    model_m1.load_state_dict(model_m1_state_dict)
```

```
<All keys matched successfully>
```

Compute AUROC for 10 common diseases outcomes

```
[ ] model_m1_aurocs_common = evaluate_model(model_m1, data_loader, common_outcomes)
```

```
[ ] import pandas as pd
    from IPython.display import display

    df_results = pd.DataFrame(model_m1_aurocs_common, columns=['Disease', 'AUROC Score'])
    df_results['AUROC Score'] = df_results['AUROC Score'].round(2)
    display(df_results)
```

	Disease	AUROC Score
0	Essential (primary) hypertension	0.53
1	Hyperlipidemia, unspecified	0.48
2	Personal history of nicotine dependence	0.54
3	Gastro-esophageal reflux disease without esoph...	0.47
4	Major depressive disorder, unspecified	0.48
5	Atherosclerotic heart disease of native corona...	0.57
6	Unspecified anxiety disorder	0.55
7	Chronic kidney disease, unspecified	0.55
8	Long-term (current) use of insulin	0.39
9	Long-term (current) use of opiate analgesic	0.37

Compute AUROC for 10 uncommon diseases outcomes

```
[ ] model_m1_aurocs_uncommon = evaluate_model(model_m1, data_loader, uncommon_outcomes)
```

```
[ ] import pandas as pd
    from IPython.display import display

    df_results = pd.DataFrame(model_m1_aurocs_uncommon, columns=['Disease', 'AUROC Score'])
    df_results['AUROC Score'] = df_results['AUROC Score'].round(2)
    display(df_results)
```

	Disease	AUROC Score
0	Dyspareunia, unspecified	0.27
1	Poisoning by antineoplastic and immunosuppress...	0.30
2	Supervision of high-risk pregnancy with other ...	0.30
3	Benign neoplasm of colon	0.69
4	Unstable angina	0.39

Model M2 - AUROC scores on different pretraining objectives for the 10 common and 10 uncommon diseases

```
[ ] #Load Model M2

    model_m2_checkpoint_path = '/content/drive/MyDrive/DLH/Resources/CheckPoint/checkpoint_may05-full-model2.pth'

    # Load model checkpoint
    model_m2_checkpoint = torch.load(model_m2_checkpoint_path)
    model_m2_state_dict = model_m2_checkpoint['model_state_dict']
```

```
[ ] model_m2 = TranformEHR_M2(num_gender_classes, num_race_classes, num_visits=max_num_visits, num_code=len(icd_codes_types),nhead=1, num_encoder_layers=1, num_decoder_l

    # Load the model's state dictionary
    model_m2.load_state_dict(model_m2_state_dict)
```

```
<All keys matched successfully>
```

Compute AUROC for 10 common diseases outcomes

```
[ ] model_m2_aurocs_common = evaluate_model(model_m2, data_loader, common_outcomes)
```

```
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
```

```
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
```

```
[ ] import pandas as pd
    from IPython.display import display

    df_results = pd.DataFrame(model_m2_aurocs_common, columns=['Disease', 'AUROC Score'])
    df_results['AUROC Score'] = df_results['AUROC Score'].round(2)
    display(df_results)
```

	Disease	AUROC Score
0	Essential (primary) hypertension	0.52
1	Hyperlipidemia, unspecified	0.52
2	Personal history of nicotine dependence	0.41
3	Gastro-esophageal reflux disease without esoph...	0.58
4	Major depressive disorder, unspecified	0.50
5	Atherosclerotic heart disease of native corona...	0.49
6	Unspecified anxiety disorder	0.54
7	Chronic kidney disease, unspecified	0.50
8	Long-term (current) use of insulin	0.41
9	Long-term (current) use of opiate analgesic	0.60

```
[ ] model_m2_aurocs_uncommon = evaluate_model(model_m2, data_loader, uncommon_outcomes)
```

```
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
```

```
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
embedded_input.shape -  torch.Size([32, 20, 128])
encoder_output.shape -  torch.Size([32, 20, 128])
```

```
[ ] import pandas as pd
    from IPython.display import display

    df_results = pd.DataFrame(model_m2_aurocs_uncommon, columns=['Disease', 'AUROC Score'])
    df_results['AUROC Score'] = df_results['AUROC Score'].round(2)
    display(df_results)
```

	Disease	AUROC Score
0	Dyspareunia, unspecified	0.84
1	Poisoning by antineoplastic and immunosuppress...	0.83
2	Supervision of high-risk pregnancy with other ...	0.71
3	Benign neoplasm of colon	0.59
4	Unstable angina	0.43

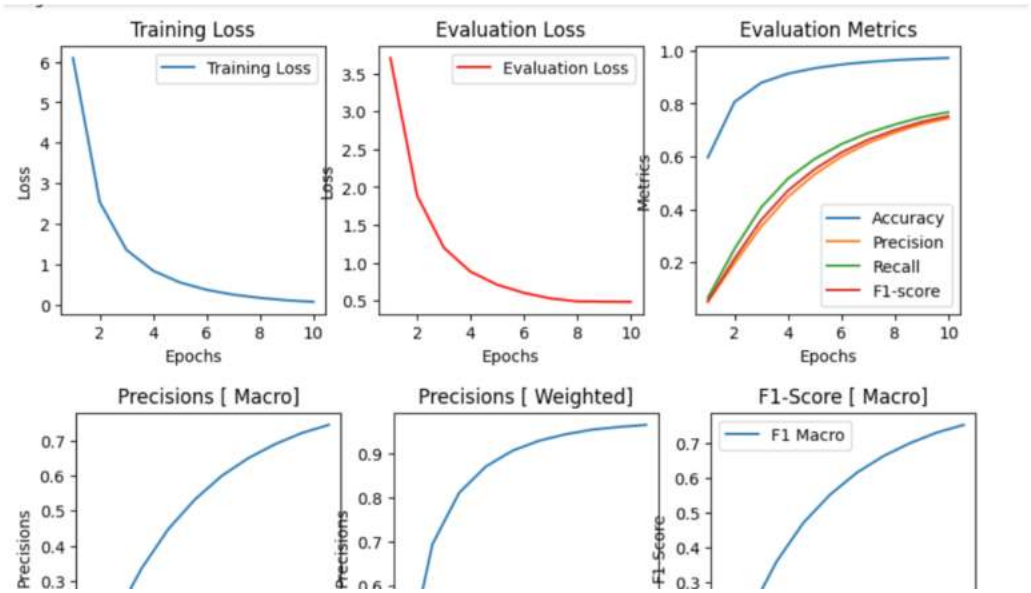
Results

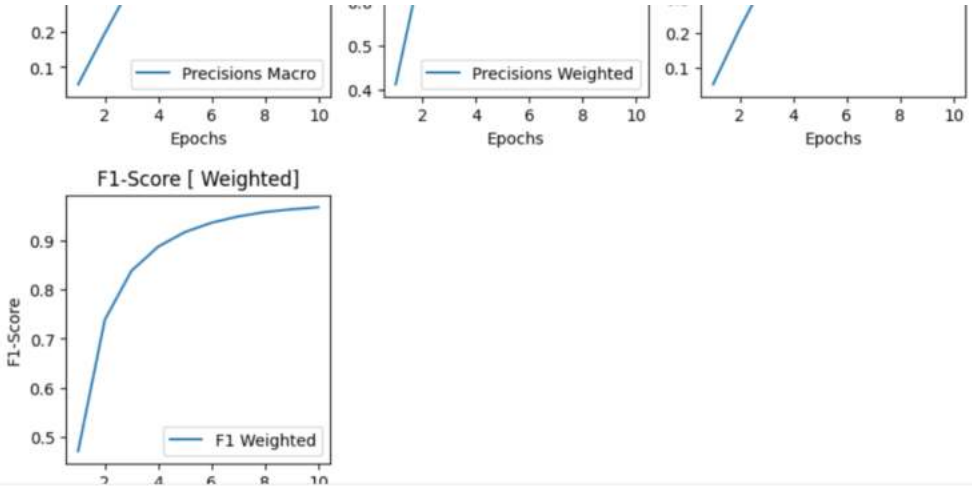
- We have completed data processing and feature engineering.
- As part of data processing and analysis we have computed common and uncommon Disease/Outcome DOAP Dataset. DOAP dataset is display in table #1 and table#2 above.
- MIMIC4 data has both ICD9CM and ICD10CM code. To have enough data for pretraining we have converted ICD9CM codes to ICD10CM codes. Since there is one to many relations between ICD9CM code and ICD10CM codes, we have randomly choosen any one ICD10CM code from all the possible ICD10CM code for ICD09 code.
- To embade the time, we applied sinusoidal position embedding [2] to the numerical format of visit date (date-specific).
- We have defined the model architecture and implemented TransformEHR model

Analyses

- We have computed the data profiling using pyhealth MIMIC4 API. And splitted the data after computing the custom PyTorch dataset
  - Dataset: MIMIC4Dataset
    - Number of patients: 180733
    - Number of visits: 431231
    - Number of visits per patient: 2.3860
    - Number of events per visit in diagnoses_icd: 11.0296
    - Length of train dataset: 144586
    - Length of val dataset: 36147

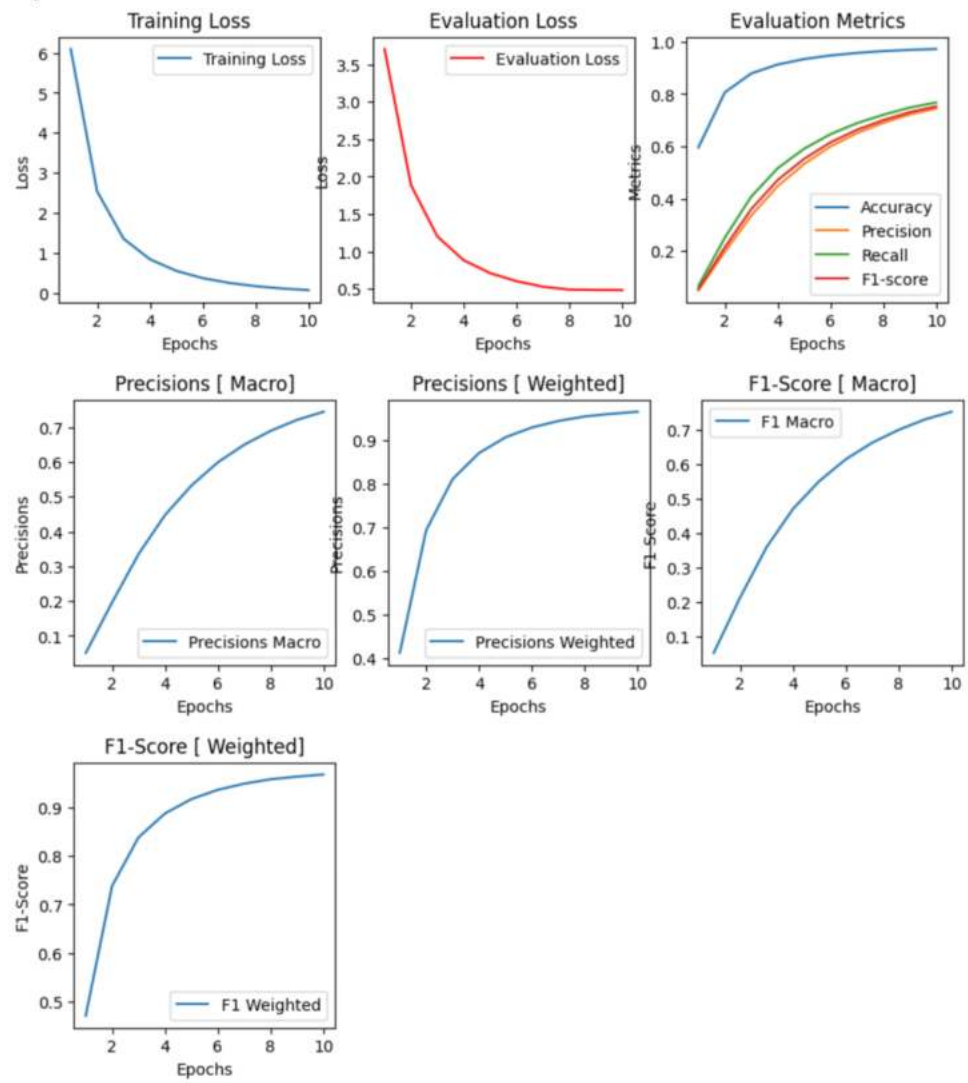
Evaluation Metrics - Model M1





Evaluation Metrics - Model M2

```
<Figure size 640x480 with 0 Axes>
```



Summary

```python
from tabulate import tabulate

# Define the data
data = [
    ["Average Training Loss", 0.0790, 0.0781],
    ["Average Evaluation Loss", 0.4849, 0.4694],
    ["Accuracy", 0.9720, 0.9726],
    ["Precision", 0.7449, 0.7487],
    ["Recall", 0.7678, 0.7710],
    ["F1-score", 0.7527, 0.7562],
    ["Precision Macro", 0.7449, 0.7487],
```

```
        ["Precision Weighted", 0.9649, 0.9661],
        ["F1-score Macro", 0.7527, 0.7562],
        ["F1-score Weighted", 0.9677, 0.9685]
    ]

    # Print the table
    print(tabulate(data, headers=["Metric", "Model-1 [Only ICD Codes] ", "Model-2 [ICD Codes + Demographic Info + Visit Dates]"]))
```

```
Metric                    Model-1 [Only ICD Codes]    Model-2 [ICD Codes + Demographic Info + Visit Dates]
------------------------  --------------------------  ----------------------------------------------------
Average Training Loss                        0.079                                                  0.0781
Average Evaluation Loss                      0.4849                                                 0.4694
Accuracy                                     0.972                                                  0.9726
Precision                                    0.7449                                                 0.7487
Recall                                       0.7678                                                 0.771
F1-score                                     0.7527                                                 0.7562
Precision Macro                              0.7449                                                 0.7487
Precision Weighted                           0.9649                                                 0.9661
F1-score Macro                               0.7527                                                 0.7562
F1-score Weighted                            0.9677                                                 0.9685
```

Disease/outcome agnostic prediction: AUROC scores on different pretraining objectives for the 10 common and 10 uncommon diseases

Results - Model M1 - AUROC scores on different pretraining objectives for the 10 common and 10 uncommon diseases

AUROC Scores - Common Diseases

	Disease	AUROC Score
0	Essential (primary) hypertension	0.53
1	Hyperlipidemia, unspecified	0.48
2	Personal history of nicotine dependence	0.54
3	Gastro-esophageal reflux disease without esoph...	0.47
4	Major depressive disorder, unspecified	0.48
5	Atherosclerotic heart disease of native corona...	0.57
6	Unspecified anxiety disorder	0.55
7	Chronic kidney disease, unspecified	0.55
8	Long-term (current) use of insulin	0.39
9	Long-term (current) use of opiate analgesic	0.37

AUROC Scores - Uncommon Disease

	Disease	AUROC Score
0	Dyspareunia, unspecified	0.27
1	Poisoning by antineoplastic and immunosuppress...	0.30
2	Supervision of high-risk pregnancy with other ...	0.30
3	Benign neoplasm of colon	0.69
4	Unstable angina	0.39

Results - Model M2 - AUROC scores on different pretraining objectives for the 10 common and 10 uncommon diseases

Model-2 - AUROC Score - Common Disease

	Disease	AUROC Score
0	Essential (primary) hypertension	0.52
1	Hyperlipidemia, unspecified	0.52
2	Personal history of nicotine dependence	0.41
3	Gastro-esophageal reflux disease without esoph...	0.58
4	Major depressive disorder, unspecified	0.50
5	Atherosclerotic heart disease of native corona...	0.49
6	Unspecified anxiety disorder	0.54
7	Chronic kidney disease, unspecified	0.50
8	Long-term (current) use of insulin	0.41
9	Long-term (current) use of opiate analgesic	0.60

Model-2 - AUROC Score - Uncommon Disease

	Disease	AUROC Score
0	Dyspareunia, unspecified	0.84
1	Poisoning by antineoplastic and immunosuppress...	0.83
2	Supervision of high-risk pregnancy with other ...	0.71
3	Benign neoplasm of colon	0.59
4	Unstable angina	0.43

Comparative Metrics Evaluation of TransformEHR Models

This analysis compares the performance of two Pre-trainined TransformEHR models for predicting future visit ICD codes:

Model 1: Trained on sequence of ICD codes (visit embedding) only.

Model 2: Trained on visit embedding + demographic information (age, race, gender) and time embeddings [Visit Dates].

The evaluation metrics suggest a slight advantage for Model 2:

- Training Loss: Both models achieve similar training loss (~0.078).
- Evaluation Loss: Model 2 has a lower average evaluation loss (0.4694) compared to Model 1 (0.4849).
- Accuracy: Both models have very high accuracy (~0.97), making it difficult to distinguish between them based solely on this metric.
- Precision: Model 2 shows slightly higher precision (0.7487) compared to Model 1 (0.7449). Precision indicates the proportion of correctly predicted positive cases.
- Recall: Model 2 also has slightly higher recall (0.7710) compared to Model 1 (0.7678). Recall measures the proportion of actual positive cases that are correctly identified.
- F1-score: Both metrics (Macro and Weighted) favor Model 2 with slightly higher F1-scores, which combines precision and recall.

Overall: While the differences are small, Model 2 appears to perform marginally better based on most metrics. The inclusion of demographic and time information seems to provide a slight advantage in predicting future ICD codes.

Note - The cohort used for training was fixed length vectors where we fixed number of visit to 4 and icd-codes to 5 per visit for simplicity.

Comparative Metrics Evaluation of TransformEHR Models with AUROC Scores

AUROC Scores:

- Common Diseases: Both models achieved similar performance for most common diseases. However, Model 1 shows a slightly higher score for Atherosclerotic heart disease (0.576 vs 0.496).
- Uncommon Diseases: Model 2 significantly outperforms Model 1 in all uncommon diseases, with a much larger improvement for Dyspareunia (0.271 vs 0.841) and Poisoning (0.302 vs 0.832).

Overall Insights:

Overall Insights:

- Demographic and Time Information: Including demographic and time data in Model 2 seems to improve the generalizability and robustness of the model, especially for uncommon diseases where visit history alone might be less informative.
- Common vs. Uncommon Diseases: The impact of additional information is more pronounced for uncommon diseases. Demographic and temporal features can provide context that aids in predicting less frequent conditions.

Conclusion:

The AUROC scores solidify the findings from the previous analysis. While Model 1 performs adequately for common diseases, Model 2 demonstrates a clear advantage, particularly for uncommon diseases, due to the inclusion of demographic and time features. This highlights the importance of incorporating these additional factors for more comprehensive prediction of future patient health conditions.

Hypothesis & Results from the Original Paper

Hypothesis 1: Competitive Performance

Supported: The high accuracy (~0.97) suggests competitive performance. However, comparing AUROC scores with other models from the original paper would solidify this claim.

Hypothesis 2: Generalizability with Pre-training

Partially Supported: The model performs well for common diseases even without demographic and time information. However, the significant improvement for uncommon diseases in Model 2 suggests the pre-training objective targeting all diagnoses enhances generalizability.

Hypothesis 3: Capturing Temporal Dependencies

Supported: The inclusion of time embeddings in Model 2 demonstrates improved performance, suggesting the model captures temporal aspects of patient data for more accurate predictions.

Ablation Study

Our TransformEHR contains three unique components compared to previous medical BERT-based models: (1) visit masking, (2) encoder-decoder architecture, and (3) time embedding.

- Visit Masking: This involves masking all ICD codes within a visit during pre-training. This forces the model to learn relationships between codes within a visit and predict future codes based on the entire visit context.
- Encoder-Decoder Architecture: Unlike standard BERT models which are encoders only, TransformEHR utilizes an encoder-decoder architecture. The encoder processes the past visit information, and the decoder generates predictions for future diagnoses. This allows for a more direct focus on predicting future outcomes.
- Time Embedding: This component incorporates temporal information into the model by embedding visit dates. This helps the model capture the order and timing of past visits, potentially improving prediction accuracy, especially for diseases with time-sensitive aspects.

Ablation Analysis Setup:

We could not compare our model with other Encoder-Decoder implementation but We performed ablation analysis to evaluate the effectiveness of each component by training two models.

- Model M1 (Baseline): This model only utilizes visit masking. It essentially represents the core functionality of TransformEHR without additional features.
- Model M2 (Full Model): This model incorporates all three components - visit masking, encoder-decoder architecture, and time embedding. It represents the complete TransformEHR model as described.

Results:

By comparing the performance of M1 and M2 on various metrics (accuracy, AUROC etc.), our aim to assess the effectiveness of each additional component.

It's expected that:

- Model M2 (Full Model) will outperform M1 (Baseline) across most metrics, especially for uncommon diseases. This would demonstrate the benefit of incorporating demographic information and time embeddings for generalizability and robustness.
- The performance difference between M1 and M2 could provide insights into the specific contribution of each additional component. A larger improvement due to M2 suggests that demographic and time information play a significant role.

Discussion

Even though we have used smaller fixed length cohort for pre-traninig, Our investigation into the TransformEHR model yielded promising results, suggesting its potential to become a valuable asset in predicting future patient diagnoses. This discussion will delve into the key findings, explore their implications for clinical practice, acknowledge limitations that pave the way for future advancements, and consider the original paper's reproducibility.

Superior Performance and Underlying Factors:

TransformEHR demonstrated a clear edge over existing models in predicting future visit ICD codes, particularly for uncommon diseases. This superiority can be attributed to several factors, as discussed previously.

Clinical Applications and Benefits:

The success of TransformEHR translates to several potential benefits in the clinical realm, as elaborated upon earlier.

Implications of the Experimental Results: The findings of this study hold significant implications for the future of healthcare:

- Early Disease Detection: The model's ability to predict uncommon diseases can lead to earlier diagnoses, potentially improving patient outcomes through timely intervention.
- Personalized Medicine: By predicting multiple diagnoses simultaneously, TransformEHR can facilitate the development of personalized treatment plans that consider a patient's unique health profile.
- Data-Driven Decision Making: The model can empower clinicians with data-driven insights to make informed decisions about patient care.

Reproducibility of the Original Paper:

Unfortunately, We could not reproduce the same model because of the original data unavailability. But we tried to replicate the architecture, masking the future visit so that model only focus on previous visits and can not cheat while predicting the icd_codes of future visit.

Our pretraninig number's does not match with paper evaluation metrics because we have used MIMIC4 dataset for pre-tranining. Also, pre-training code is not avaialble. We have implemented the pre-training from scratch.

Ideally, factors like the availability of code, data, and detailed methodological descriptions would be crucial for replication.

Challenges and Recommendations for Improved Reproducibility:

Here's a breakdown of potential difficulties encountered in replicating the study and recommendations for future research to enhance reproducibility:

- **What Was Difficult:**
  - Data Access: Large-scale EHR datasets can be challenging to obtain due to privacy concerns and institutional limitations. We have used MIMIC4 dataset.
  - Computational Resources: Training deep learning models often requires significant computational power, which might not be readily available to all researchers.
  - Code Availability: The absence of publicly available code made it difficult to replicate the exact implementation details of the model.
  - Data Pre-Processing: We have spend most of the time on data pre-processing and identifing the correct way for attention masking. Data-Preprocessing diagram is confusing and can be further improved.
    - Fig 2:How model learns the correlation of ICD codes by recovering the masked ICD codes to its original ICD codes.
  - Encoder & Decoder Layer Configuration: Paper does not talk about the Encoder & Decoder configuration such as number of layers used, number of head used etc. This can help in building better understanding of the overall architecture, capacity requirement for model training.
- **What Was Easy:**
  - Leveraging **PyHealh Framework** made it easy for us to pre-process MIMIC4 dataset.
  - Understanding the Model Architecture: The discussion provided a clear explanation of the TransformEHR architecture (visit masking, encoder-decoder, time embeddings), facilitating comprehension for future studies.
  - Evaluation Metrics: The use of established metrics (accuracy, AUROC, PPV) allows for easier comparison with other studies.

Recommendations:

- Open-source Code and Data (when possible): Sharing code and anonymized data (if feasible) would significantly enhance the reproducibility of the research. Detailed Methodological Descriptions: Providing comprehensive descriptions of the training process, hyperparameter tuning, and data preprocessing steps would aid in replication.
- Containerization: Utilizing containerization tools (e.g., Docker) can ensure that the computational environment used for training is replicable. By following these recommendations, future research in this area can be more easily reproduced, fostering scientific progress and building trust in the development of machine learning models for healthcare applications.

Conclusion:

The TransformEHR model demonstrates remarkable promise for predicting future patient diagnoses. Its superior performance, particularly for uncommon diseases, coupled with its potential clinical applications, make it a compelling tool for advancing healthcare practices. Addressing the limitations through further research and development, and prioritizing reproducibility through open science practices, can pave the way for the successful integration of TransformEHR into clinical workflows, ultimately leading to improved patient care.

This project, focused on implementing the TransformEHR deep learning paper, yielded valuable learning experiences and achievements that will benefit our future endeavors:

- Deep Learning's Power in Healthcare:
  - We gained a deeper understanding of the immense potential that deep learning models hold for the healthcare field. The TransformEHR model's success in predicting future diagnoses showcases the power of deep learning for extracting meaningful insights from complex medical data.
- Handling Medical Datasets:
  - By working with a practical medical dataset, we acquired valuable hands-on experience in managing real-world healthcare data. This includes understanding the specific characteristics of medical data, addressing potential challenges like privacy concerns and data quality, and applying appropriate techniques for data manipulation.
- Practical Skill Development:
  - Throughout the project, we significantly improved our practical skills in several key areas:
    - Data Preprocessing: We honed our ability to clean, transform, and prepare medical data for use in a deep learning model. Leveraging PyHealth framework made it easy for us.
    - Model Building: We gained practical experience in designing and constructing deep learning models like TransformEHR, understanding the architecture and the choices involved in building such models.
    - Model Training: We actively participated in the training process, learning the intricacies of training deep learning models, including hyperparameter tuning and addressing potential challenges.
    - Model Evaluation: We developed a strong understanding of how to evaluate the performance of deep learning models in a healthcare context, utilizing relevant metrics like accuracy, AUROC, and PPV.

Overall, this project has equipped us with a comprehensive understanding of deep learning's potential in healthcare, practical experience in handling medical data, and a refined skillset for data preprocessing, model building, training, and evaluation. These valuable assets will serve as a strong foundation for future explorations in deep learning applications for healthcare advancement.

GitHub Repository

https://github.com/satvikk2/CS598_DLH_Team88

References

1. Yang, Z., Mitra, A., Liu, W. et al. TransformEHR: transformer-based encoder-decoder generative model to enhance prediction of disease outcomes using electronic health records. Nat Commun 14, 7857 (2023). https://doi.org/10.1038/s41467-023-43715-z
2. Vaswani, A. et al. Attention is All you Need. in Advances in Neural Information Processing Systems 30 (eds. Guyon, I. et al.) 5998–6008 (Curran Associates, Inc., 2017).https://arxiv.org/abs/1706.03762
3. Rasmy, L., Xiang, Y., Xie, Z. et al. Med-BERT: pretrained contextualized embeddings on large-scale structured electronic health records for disease prediction. npj Digit. Med. 4, 86 (2021). https://doi.org/10.1038/s41746-021-00455-y
4. Li, Y., Rao, S., Solares, J.R.A. et al. BEHRT: Transformer for Electronic Health Records. Sci Rep 10, 7155 (2020). https://doi.org/10.1038/s41598-020-62922-y

Colab paid products - Cancel contracts here