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

CS598 Project Draft

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

GitHub Repository

https://github.com/satvikk2/CS598_DLH_Team88

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.

We can add more details inlcuding architecture here.

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.

Data

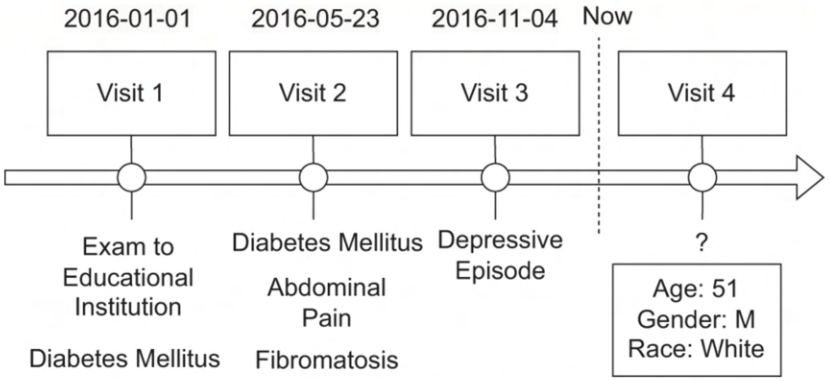
The dataset we plan to use in this project is MIMIC-IV from https://physionet.org. The MIMIC-IV dataset includes intensive care unit patients admitted to the Beth Israel Deaconess Medical Center in Boston, Massachusetts, comprises deidentified patient records used for medical research and analysis. It 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
  - Admitions,
  - 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

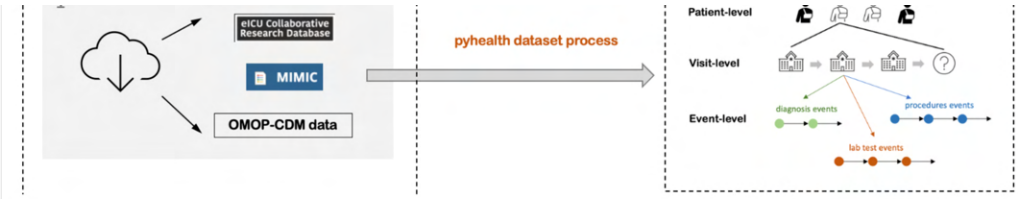
Data Processing (feature engineering)

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

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

```
[ ] # import  packages you need
    import numpy as np
    import torch
    import torch.nn as nn
    import torch.nn.functional as F
```

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

```
[3] """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
```

```
    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"""
```

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)"""
```

```
['10000032', '10000068', '10000084', '10000108', '10000117', '10000248', '10000280', '10000560', '10000635', '10000719']
Patient 10000032 with 4 visits:
        - Birth datetime: 2128-04-07 00:00:00
        - Death datetime: 2180-09-09 00:00:00
        - Gender: F
        - Ethnicity: WHITE
        - anchor_year_group: 2014 - 2016
        - Visit 22595853 from patient 10000032 with 8 events:
                - Encounter time: 2180-05-06 22:23:00
                - Discharge time: 2180-05-07 17:15:00
                - Discharge status: 0
                - Available tables: ['diagnoses_icd']
                - Event from patient 10000032 visit 22595853:
                        - Code: 5723
                        - Table: diagnoses_icd
                        - Vocabulary: ICD9CM
                        - Timestamp: None
                - Event from patient 10000032 visit 22595853:
                        - Code: 78959
                        - Table: diagnoses_icd
                        - Vocabulary: ICD9CM
                        - Timestamp: None
                - Event from patient 10000032 visit 22595853:
                        - Code: 5715
                        - Table: diagnoses_icd
                        - Vocabulary: ICD9CM
                        - Timestamp: None
                - Event from patient 10000032 visit 22595853:
                        - Code: 07070
                        - Table: diagnoses_icd
                        - Vocabulary: ICD9CM
                        - Timestamp: None
                - Event from patient 10000032 visit 22595853:
                        - Code: 496
                        - Table: diagnoses_icd
                        - Vocabulary: ICD9CM
                        - Timestamp: None
                - Event from patient 10000032 visit 22595853:
                        - Code: 29680
                        - Table: diagnoses_icd
                        - Vocabulary: ICD9CM
                        - Timestamp: None
                - Event from patient 10000032 visit 22595853:
                        - Code: 30981
                        - Table: diagnoses_icd
                        - Vocabulary: ICD9CM
                        - Timestamp: None
```

```
                - Event from patient 10000032 visit 22595853:
                        - Code: V1582
                        - Table: diagnoses_icd
                        - Vocabulary: ICD9CM
                        - Timestamp: None
        - Visit 22841357 from patient 10000032 with 8 events:
                - Encounter time: 2180-06-26 18:27:00
                - Discharge time: 2180-06-27 18:49:00
                - Discharge status: 0
                - Available tables: ['diagnoses_icd']
                - Event from patient 10000032 visit 22841357:
```

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()

    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))"""
```

```
10
Selected Common ICD-10CM Codes: ['I10', 'E785', 'Z87891', 'K219', 'F329', 'I2510', 'F419', 'N179', 'Z794', 'Z7901']
Selected Uncommon ICD-10CM Codes: ['N94.6', 'T47.1X5D', 'O30.033', 'I70234', 'I95.2', 'Z34.83', 'C8518', 'L89.891', 'D126', 'I201']
Number of patients in Common Disease/Outcome DOAP Dataset: 133711
Number of patients in Uncommon Disease/Outcome DOAP Dataset: 340
```

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
```

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
```

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.

```
[5] # 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
      print(birth_date, death_date)
      age = death_date.year - birth_date.year - ((death_date.month, death_date.day) < (birth_date.month, birth_date.day))
      return age

    types = {}
    gender2idx = {}
    race2idx = {}
    age = []
    gender = []
    race = []
    visit_dates = []

    def transform_ehr_mimic4_fn(patient):
        """
        types = {}
      newSeqs = []
      for patient in seqs:
        newPatient = []
        for visit in patient:
          newVisit = []
          for code in visit:
            if code in types:
              newVisit.append(types[code])
            else:
              types[code] = len(types)
              newVisit.append(types[code])
          newPatient.append(newVisit)
        newSeqs.append(newPatient)

        """
        newPatient = []
        visit_date = []

        for i in range(len(patient)):

          visit = patient[i]
          formatted_visit_date = visit.encounter_time.strftime("%Y-%m-%d")
          visit_date.append(formatted_visit_date)

          conditions = []

          events = visit.get_event_list(table="diagnoses_icd")
          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(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])

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

        p_ethnicity = patient.ethnicity
        if p_ethnicity in race2idx:
          race.append(race2idx[p_ethnicity])
        else:
          race2idx[p_ethnicity] = len(race2idx)
          race.append(race2idx[p_ethnicity])
        return newPatient
```

Ordering Visits Based on Visit Dates

```
[ ] # sort the visit_date and seqs based on the visit date
    sorted_seqs = []
    sorted_visit_dates = []
    for i in range(len(seqs)):
        visit_date = 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
    visit_dates = sorted_visit_dates
    print(seqs[0])
    print(visit_dates[0])
```

```
[ ] print(visit_dates[0])
```

```
    ['2180-05-06', '2180-06-26', '2180-08-05', '2180-07-23']
```

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 Drive.

```
[ ] import pickle

    # Assuming your data object is named 'data_object'
    mimic4_ds_seqs_path = '/content/drive/MyDrive/DLH/MIMIC4/PKL/seqs.pkl'
    mimic4_ds_visit_dates_path = '/content/drive/MyDrive/DLH/MIMIC4/PKL/dates.pkl'
    mimic4_ds_type_path = '/content/drive/MyDrive/DLH/MIMIC4/PKL/type.pkl'
    mimic4_ds_gender_path = '/content/drive/MyDrive/DLH/MIMIC4/PKL/gender.pkl'
    mimic4_ds_race_path = '/content/drive/MyDrive/DLH/MIMIC4/PKL/race.pkl'
    mimic4_ds_age_path = '/content/drive/MyDrive/DLH/MIMIC4/PKL/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_visit_dates_path, 'wb') as f:
        pickle.dump(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(gender, f)

    with open(mimic4_ds_race_path, 'wb') as f:
```

```
        pickle.dump(race, f)

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

Load Data for Model Training

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

    # Path to the saved data object
    # Assuming your data object is named 'data_object'
    mimic4_ds_seqs_path = '/content/drive/MyDrive/DLH/MIMIC4/PKL/seqs.pkl'
    mimic4_ds_visit_dates_path = '/content/drive/MyDrive/DLH/MIMIC4/PKL/dates.pkl'
    mimic4_ds_type_path = '/content/drive/MyDrive/DLH/MIMIC4/PKL/type.pkl'
    mimic4_ds_gender_path = '/content/drive/MyDrive/DLH/MIMIC4/PKL/gender.pkl'
    mimic4_ds_race_path = '/content/drive/MyDrive/DLH/MIMIC4/PKL/race.pkl'
    mimic4_ds_age_path = '/content/drive/MyDrive/DLH/MIMIC4/PKL/age.pkl'

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

```
[ ] print(len(seqs))
    print(len(visit_dates))
    print(len(gender))
    print(len(race))
    print(len(age))
    print(len(icd_codes_types))
```

```
180733
180733
180733
180733
180733
71273
```

Build The Dataset

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

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

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

    class CustomDataset(Dataset):
      def __init__(self, seqs, gender, race, age, visit_dates):
        self.x = seqs
        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]
        gender = self.gender[index]
        race = self.race[index]
        age = self.age[index]
        visit_dates = self.visit_dates[index]
        # Return the pair (sequence, hf)
        print(sequence, gender, race, age, visit_dates)
        return (sequence, gender, race, age, visit_dates)

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

```
[ ] print(dataset.__getitem__(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]] 0 0 52 ['21
([[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]], 0, 0, 52,
```

Split Data Into Train and Validation Set

Now we have CustomDataset. Let us split the dataset into training and validation sets.

```
[ ] 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: 144586
Length of val dataset: 36147
```

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)

```
[ ] from torch.utils.data import DataLoader
    import math

    def load_data(dataset, batch_size):
      def collate_fn(data):
        """
        Arguments:
            data: a list of samples fetched from `CustomDataset`
        Outputs:
            x: a tensor of shape (# patiens, max # visits, max # diagnosis codes) of type torch.long
            masks: a tensor of shape (# patiens, max # visits, max # diagnosis codes) of type torch.bool
            gender: a tensor of shape (# patients) of type torch.long (optional)
            race: a tensor of shape (# patients) of type torch.long (optional)
            visit_dates: a list of lists of strings representing visit dates (optional)
        """
        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))
          pe[0::2] = torch.sin(position * div_term)
          pe[1::2] = torch.cos(position * div_term)
          return pe.unsqueeze(0)

        sequences, 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)

        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_codes = max(num_codes)
        x = torch.zeros((num_patients, max_num_visits, max_num_codes), dtype=torch.long)
        masks = torch.zeros((num_patients, max_num_visits, max_num_codes), 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)):
          for j_visit, visit in enumerate(patient):
            # Mask all ICD codes in the visit
            masked_visit = [0] * len(visit)  # Replace with actual masking logic (e.g., random masking)
            padded_seq = torch.tensor(masked_visit + [0] * (max_num_codes - len(visit)), dtype=torch.long)
            x[i_patient, j_visit, :] = padded_seq
            masks[i_patient, j_visit, :] = torch.ones(max_num_codes, dtype=torch.bool)  # All codes masked in this visit
            # 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, masks, gender, race, age, position_encodings)

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

```
[ ] 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(val_loader)
    x, masks, gender, race, age, position_encodings = next(loader_iter)
    print(x, masks, gender, race, age, position_encodings)
```

```
[ ] print("x", x.shape)
    print("masks", masks.shape)
    print("gender", gender.shape)
    print("race", race.shape)
    print("age", age.shape)
    print("position_encodings", position_encodings.shape)
```

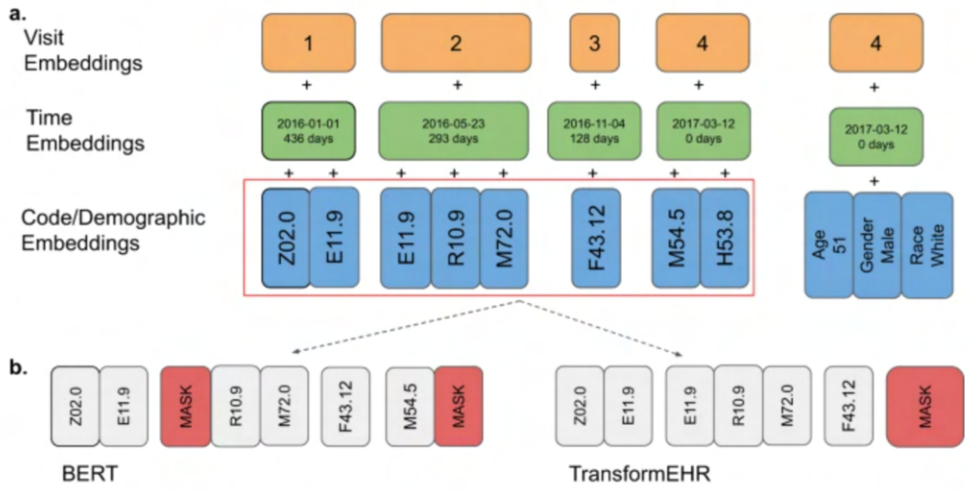
```
x torch.Size([32, 10, 30])
masks torch.Size([32, 10, 30])
gender torch.Size([32])
race torch.Size([32])
age torch.Size([32])
position_encodings torch.Size([32, 10, 2])
```

position_encodings torch.Size([32, 10, 2])

Model

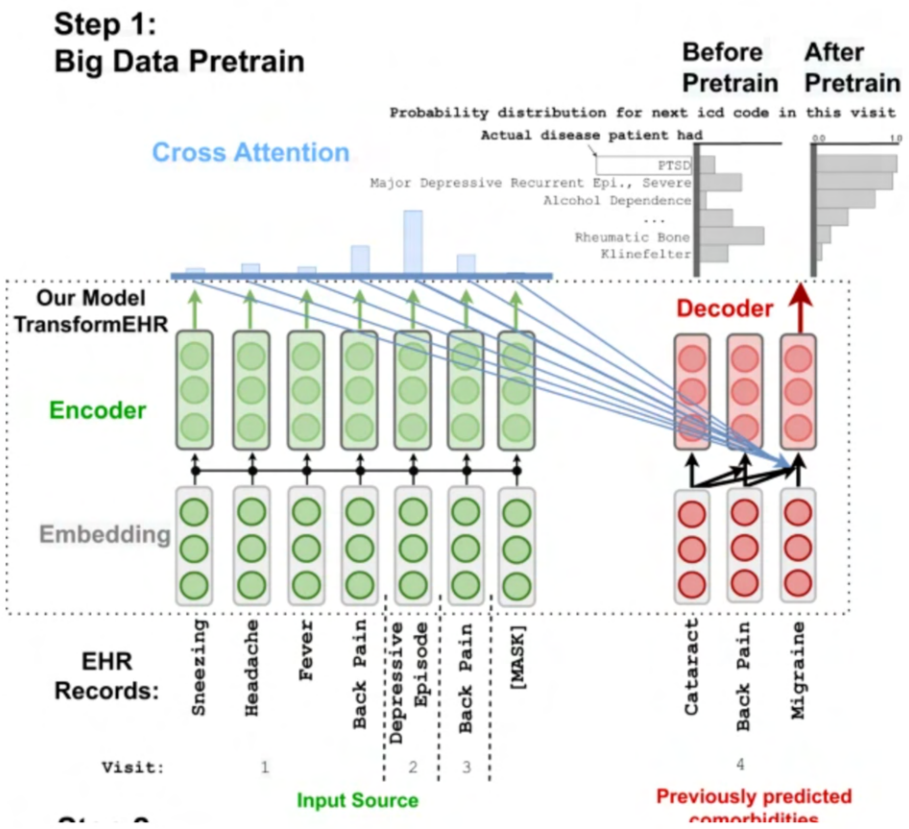
TransformEHR Model Architecture

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.

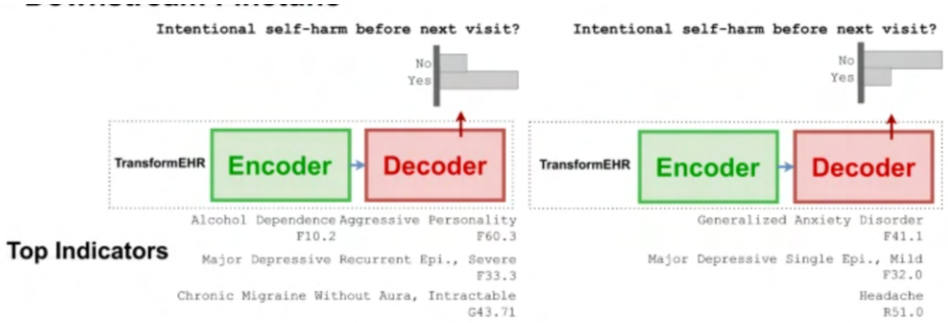


a illustrates the preprocessing of input data from 4 patient visits, with visit embeddings capturing chronological information, time embeddings capturing the specific date, and ICD codes along with demographic data encoded as embeddings. **b** displays various medical history masking schemes: BERT-style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

```python
# 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
max_num_visits = 18
max_num_codes = 35

class TranformEHR(nn.Module):
  def __init__(self, num_gender_classes, num_race_classes, num_code, nhead, num_encoder_layers, num_decoder_layers, embedding_dim=128):
    super().__init__()
    # HZ
    self.embedding_dim = embedding_dim
    self.concatenated_dim = embedding_dim * 5
    self.projected_dim = embedding_dim


    # Define the embedding layers
    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)

    # HZ
    # self.embedding_projection = nn.Linear(embedding_dim * 5, self.projected_dim)
    self.embedding_projection = nn.Linear(self.concatenated_dim, self.projected_dim)

    # Transformer encoder
    encoder_layer = nn.TransformerEncoderLayer(d_model=embedding_dim, nhead=nhead)
    self.transformer_encoder = nn.TransformerEncoder(encoder_layer, num_layers=num_encoder_layers)
    # Transformer decoder
    # decoder_layer = nn.TransformerDecoderLayer(d_model=embedding_dim, nhead=nhead)
    decoder_layer = nn.TransformerDecoderLayer(d_model=self.projected_dim, nhead=nhead)
```

```
        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(embedding_dim, num_code)
        self.linear = nn.Linear(self.projected_dim, num_code)

      def forward(self, x, masks, gender, race, age, position_encodings):
        embedded_gender = self.gender_embedding(gender)
        embedded_race = self.race_embedding(race)
        #embedded_age = self.age_embedding(age.float())
        # HZ
        # embedded_age = self.age_embedding(age.view(-1, 1).float())
        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)

        # Concatenate all embeddings
        print(embedded_x.shape)
        print(embedded_positional_encodings.shape)
        print(embedded_age.shape)
        print(embedded_race.shape)
        print(embedded_gender.shape)
        # HZ
        # Add dimensions to positional encodings and other embeddings
        # embedded_positional_encodings = embedded_positional_encodings.unsqueeze(2).expand(-1, -1, 27, -1)
        # embedded_age = embedded_age.unsqueeze(1).unsqueeze(1).expand(-1, 13, 27, -1)
        # embedded_race = embedded_race.unsqueeze(1).unsqueeze(1).expand(-1, 13, 27, -1)
        # embedded_gender = embedded_gender.unsqueeze(1).unsqueeze(1).expand(-1, 13, 27, -1)
        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)

        # print("Shape of embedded_x:", embedded_x.shape)
        # print("Shape of embedded_positional_encodings:", embedded_positional_encodings.shape)
        # print("Shape of embedded_age:", embedded_age.shape)
        # print("Shape of embedded_race:", embedded_race.shape)
        # print("Shape of embedded_gender:", embedded_gender.shape)
        embedded_input = torch.cat((embedded_x, embedded_positional_encodings, embedded_age, embedded_race, embedded_gender), dim=-1)
        # HZ
        embedded_input = self.embedding_projection(embedded_input)

        print(embedded_input.shape)
        # HZ
        # reshaped_input = embedded_input.reshape(32, 13*27, -1)
        reshaped_input = embedded_input.reshape(embedded_input.size(0), -1, self.projected_dim)
        print(reshaped_input.shape)
        # Apply transformer encoder
        print("shape of masks: ", masks.shape)
        # HZ
        # reshaped_masks = masks.reshape(masks.size(0), -1)
        if masks is not None:
            reshaped_masks = masks.view(masks.size(0), -1)

        # Confirm sizes match
        print(f"Input size: {embedded_input.shape}")  # Expected [batch_size, seq_len, features]
        print(f"Mask size: {reshaped_masks.shape}")   # Expected [batch_size, seq_len]

        # encoder_output = self.transformer_encoder(reshaped_input, src_key_padding_mask=masks.reshape(32, 13*27))
        encoder_output = self.transformer_encoder(reshaped_input, src_key_padding_mask= reshaped_masks)

        # Apply transformer decoder
        decoder_output = self.transformer_decoder(embedded_input, encoder_output, tgt_key_padding_mask=masks)

        # Project decoder output to ICD code probabilities
        logits = self.linear(decoder_output)

        return logits

    # load the model here
    model = TranformEHR(num_gender_classes, num_race_classes, num_code=len(icd_codes_types), nhead=2, num_encoder_layers=1, num_decoder_layers=1)
```

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 2 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.

```python
import torch
from torch import nn
from torch.utils.data import DataLoader

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

# Define your optimizer (e.g., Adam)
optimizer = torch.optim.Adam(model.parameters())

log_interval = 5

def train(model, train_data_loader, epochs):
  """
  Train the TransformEHR model on the provided dataloader.

  Args:
      model (nn.Module): The TransformEHR model to train.
      dataloader (DataLoader): The dataloader containing training data.
      epochs (int): Number of training epochs.
  """
  model.train()  # Set model to training mode
  for epoch in range(epochs):
    for batch_idx, batch in enumerate(train_loader):
      x, masks, gender, race, age, position_encodings = batch
      # Move data to the device (GPU if available)
      #x, masks, gender, race, age, position_encodings, labels = x.to(device), masks.to(device), gender.to(device), race.to(device), age.to(device), position_encoc
      # Forward pass
      logits = model(x, masks, 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()

      # Print training progress (optional)
      if (i + 1) % log_interval == 0:
        print(f"Epoch [{epoch+1}/{epochs}], Step [{i+1}/{len(train_data_loader)}], Loss: {loss.item():.4f}")

def eval(model, val_data_loader):
  """
  Evaluate the TransformEHR model on the provided dataloader.

  Args:
      model (nn.Module): The TransformEHR model to evaluate.
      dataloader (DataLoader): The dataloader containing evaluation data.

  Returns:
      float: Average loss on the evaluation data
  """
  model.eval()  # Set model to evaluation mode
  with torch.no_grad():  # Disable gradient calculation for efficiency
    total_loss = 0
    for x, masks, gender, race, age, position_encodings in val_data_loader:
      # Move data to the device (GPU if available)
      #x, masks, gender, race, age, position_encodings, labels = x.to(device), masks.to(device), gender.to(device), race.to(device), age.to(device), position_encoc

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

    # Calculate average loss
    avg_loss = total_loss / len(val_data_loader)
    return avg_loss

# Example usage
train(model, train_loader, 2)
eval_loss = eval(model, val_loader)
print(f"Evaluation Loss: {eval_loss:.4f}")
```

```
[[2220, 408, 16443, 3668, 889, 462, 1135, 75, 40, 12797, 3664, 283, 739, 403, 2145, 2462, 439], [409, 3627, 2746, 435, 22616, 889, 462, 75, 532, 40, 16443, 63, 283,
[[3712, 16388, 19684, 5158, 9731, 3010, 17851, 3814, 4354, 1161, 1437, 31921, 5955, 7372, 8632]] 0 0 70 ['2122-06-16']
[[531, 4761], [3040, 666, 1595, 582, 11313, 229, 12582, 1175, 585, 751], [26203, 15609, 8591, 8824, 783, 2049, 22450, 1541, 75, 582, 7206, 156, 684], [383, 75, 404
[[7177, 113, 32891, 545, 9203, 903]] 0 8 23 ['2150-12-17']
[[2662, 3814, 2247, 553]] 0 0 32 ['2135-06-27']
[[7731, 1416, 4354, 22107], [20882, 16016, 395, 5211, 25461]] 0 0 33 ['2158-11-06', '2167-02-18']
[[2122, 11859, 4093, 11756, 7036, 1125, 2885, 4361, 31807, 10453, 252, 13398, 4152, 6327, 531, 778, 7161, 8795, 20, 5033, 12905, 2546, 4193]] 0 0 75 ['2115-04-14']
[[14193, 97], [24820, 6989]] 0 9 37 ['2184-06-24', '2187-01-10']
[[559, 939, 204, 616, 7794, 512, 622, 55, 40, 538, 41, 927, 5791, 432, 3266]] 1 1 76 ['2144-02-10']
[[1401, 75, 395, 214, 55, 3753, 222, 8114], [4732, 16743, 13382, 38337, 52913, 718, 49681, 4760, 75, 6928]] 0 1 57 ['2166-08-24', '2160-11-01']
[[1076, 1488, 488, 75, 532, 801, 3116, 38442, 398, 52, 3013, 891, 6115], [10521, 17180, 2837, 628, 488], [1076, 38442, 1082, 488, 75, 801]] 1 21 50 ['2165-03-13', '
[[3259, 8265, 829, 9131, 1982, 1333, 1308, 3864, 414, 8929, 399, 3543, 3264]] 1 0 66 ['2164-05-10']
[[13308, 75, 13819, 40, 14, 13726, 222, 3516]] 0 4 64 ['2152-08-21']
[[56973, 252, 36041, 16830, 4875, 56974, 75, 4438, 6541, 56975, 8632, 1626, 11, 2507]] 1 4 68 ['2167-03-20']
[[2055, 2496, 50]] 0 0 53 ['2196-12-10']
[[2022, 14785, 674, 2616, 6397, 147, 4595, 1643, 3485, 8357, 5944, 664, 252, 260, 298, 250, 1441, 7301, 2876, 255, 296, 2302, 769, 1623, 11952, 2070, 474, 9670, 148
[[23004, 83, 24108, 2616, 9584, 2893, 24296, 230, 147]] 1 2 50 ['2156-05-11']
[[55989, 4042, 4791, 516, 30877, 54518, 296, 75]] 0 17 65 ['2132-05-08']
[[4947, 856, 9250, 3153, 633, 317, 224, 21156, 260, 10423, 1384, 2556, 1920, 16003, 2224, 91, 7612, 17750, 710, 1687, 12573, 15, 660, 240, 25222], [92, 92, 1680, 49
[[5833, 9469, 763, 160, 11756, 681, 2012, 786, 203, 5007, 236], [101, 23117, 292, 18478, 18900, 638, 1885, 1717, 871, 305, 174], [17804, 13482, 12215, 261, 75, 633
[[5471, 11800, 6729, 9548, 1705, 10980, 2494, 100, 2013, 5531, 3276, 154, 8653, 1671, 148], [901, 5379, 322, 14465, 5567, 154]] 1 0 52 ['2171-07-16', '2171-07-04']
[[574, 252, 465, 40, 17893, 55, 1077, 739]] 1 0 87 ['2160-05-13']
[[716, 664, 780, 305, 257, 174, 262, 1251, 8795, 6465], [498, 678, 1655, 1420, 1385, 4992, 711, 3563, 973, 637, 6903, 3160, 2391, 1415], [13715, 1204, 75, 255, 653,
```

```
[[65182, 4099, 57572, 12554, 10442, 22442], [10223, 18657, 12849]] 0 27 19 ['2173-08-04', '2170-09-30']
[[61227, 14690, 34939, 32796, 2859, 2057, 2747, 1789, 2508, 6966, 24261, 1062, 1773, 20317, 811, 1982, 1488, 17786, 285, 1916, 2051]] 1 2 60 ['2110-04-01']
[[4251, 5275, 22750, 4458, 1439, 127, 75, 1175, 130, 11156], [388, 12834, 75, 1782, 513, 577, 534, 4495, 214, 63, 414]] 0 0 23 ['2116-03-03', '2116-09-28']
[[24702, 7167, 34634, 625, 52, 55, 214]] 0 0 43 ['2168-05-24']
[[14082, 1514, 2967, 532, 211, 625, 75, 9988, 14157, 3742, 395, 536, 37556, 1820, 50, 54, 55, 877, 222, 2872, 8440, 5967, 534, 1061, 1297, 1775], [32260, 14530, 879
[[714, 2441, 2443, 2326, 3397, 294, 1441, 1140, 527, 5446, 6519, 11669, 125, 9308, 14567, 3804, 6103, 3636, 17937, 658, 383, 5310]] 1 0 63 ['2172-11-27']
[[33037, 16240, 7506, 4685, 3781, 904, 1057, 4678, 1805], [5670], [13366, 1947, 75, 183, 3701, 11144]] 0 2 69 ['2157-11-18', '2157-12-29', '2156-08-14']
[[6469, 1514, 206, 214, 889, 697, 63, 282, 47, 425, 432, 440, 1789, 417, 1319, 1309, 738], [8573, 1514, 1562, 1294, 925, 1309, 29326, 412, 878, 889, 214, 439, 1789,
[[3054, 75, 229]] 0 0 66 ['2169-05-31']
torch.Size([32, 9, 32, 128])
torch.Size([32, 9, 128])
torch.Size([32, 128])
torch.Size([32, 128])
torch.Size([32, 128])
torch.Size([32, 9, 32, 128])
torch.Size([32, 288, 128])
shape of masks:  torch.Size([32, 9, 32])
Input size: torch.Size([32, 9, 32, 128])
Mask size: torch.Size([32, 288])
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-22-a398d3fb4ced> in <cell line: 69>()
     67
     68 # Example usage
---> 69 train(model, train_loader, 2)
     70 eval_loss = eval(model, val_loader)
     71 print(f"Evaluation Loss: {eval_loss:.4f}")

                                 14 frames
/usr/local/lib/python3.10/dist-packages/torch/nn/functional.py in multi_head_attention_forward(query, key, value, embed_dim_to_check, num_heads, in_proj_weight,
in_proj_bias, bias_k, bias_v, add_zero_attn, dropout_p, out_proj_weight, out_proj_bias, training, key_padding_mask, need_weights, attn_mask,
use_separate_proj_weight, q_proj_weight, k_proj_weight, v_proj_weight, static_k, static_v, average_attn_weights, is_causal)
   5413     # merge key padding and attention masks
   5414     if key_padding_mask is not None:
-> 5415         assert key_padding_mask.shape == (bsz, src_len), \
   5416             f"expecting key_padding_mask shape of {(bsz, src_len)}, but got {key_padding_mask.shape}"
   5417         key_padding_mask = key_padding_mask.view(bsz, 1, 1, src_len).   \

AssertionError: expecting key_padding_mask shape of (288, 32), but got torch.Size([32, 288])
```

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

Plans

- Since the original code is not available, we are implementing the paper from scratch using PyHealth , PyTorch Transformer API. As of now we are working on Model Training and Eveluation component.
  - Next Action Items
    - Complete pretrained model training and eveluation.
    - Eveluate the pretrained model on DOAP datasets
    - Implement Step#2 finetune on two diseases/outcomes.
    - Prepare the eveluation figures
    - If time permits, we will also work on model comparisons and ablations study.

```
[ ] # metrics to evaluate my model

    # plot figures to better show the results

    # it is better to save the numbers and figures for your presentation.
```

Model comparison - TBD

```
[ ] #TBD
```

Discussion - TBD

```
[ ] # no code is required for this section
    '''
    if you want to use an image outside this notebook for explanaition,
    you can read and plot it here like the Scope of Reproducibility
    '''
```

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