Geometric Deep Learning for Clinical Decision Support

Candidate number: 1045182

Word count: 7496 (overleaf)
not including appendix code after bibliography
which starts on page 43

University of Oxford

A thesis submitted for the degree of
Master of Mathematics and Computer Science

Trinity 2023

Acknowledgements

It is difficult to write a heartfelt acknowledgement while preserving my anonymity, but here goes:

thanks mum.

Contents

List of Figures

Chapter 1

Introduction

During a hospital stay, patients generate large amounts of data. Like many fields, medical records are moving towards digitisation. Not only are electronic data available anywhere in the country, simplifying patient care and ensuring proper care in an emergency, but they are also pre-processed by the software used to record them. The records are stored in a well-structured manner, typically relational databases. This processing means that data science is much more doable now on medical data than previously.

Access to electronic health records (henceforth EHRs) is meant to help healthcare professionals make better-informed decisions about patient care. Access to a patient's health records may aid that individual's treatment, but this information cannot easily be used to help other patients. Likewise, the EHR may show an individual's previous responses to treatments, but cannot draw on the experiences of similar patients receiving similar treatments. It is natural to ask whether AI models with access to EHRs can improve patient care, with a global view of the data unavailable to doctors.

Medical data, and graph data generally, is difficult to use as it varies in size and type - early machine learning (ML) models take a fixed-size input, of only one datatype. However, over the last decade, ML models have been successful on a variety of datasets, across medical and non-medical fields, e.g. social media algorithms [40], traffic prediction [18], drug discovery [36], medical imaging analysis [25]. These models have been trained on, and learned from, vast amounts of data - far more than any human brain could process, and have very little input from experts in the field. Inspired by these successes, researchers have been trying to apply ML to the healthcare domain to solve a number of canonical problems, including prediction of readmission [3], morbidity [29], and length of stay (LoS) [8]. These works are promising, but their approaches are somewhat disparate, due partly to the rich and complex structure of medical data. Often, these models are useful for predicting

individual traits, but don't leverage the full power of comparing similar individuals to make more accurate assessments.

Geometric Deep Learning (GDL) has been developed as a method to unify and derive ML architectures that respect underlying data structure. In the same way that convolutional neural networks [31] "make sense" for image analysis, as their architecture interacts "properly" with shift-invariance in images, graph neural networks (GNNs) have been developed as the models for deep learning on graph-structured data. The relational databases holding the EHR data are easily translated into a graph-like form, and we can therefore analyse the structure of the data to inform our architecture.

This dissertation investigates learning on large-batch medical data, and proves results about heterogeneous architecture, using a novel application of the GDL framework to heterogeneous graphs. The later experiment attempts analysis similar to [19] and [15]. These use discharge summaries to predict patient readmission - instead I use graphs derived from patient data. I attempt automatic inference of patient readmission using information from their hospital stay, without any input from a clinician. I first give necessary background on GDL and Graph ML, and then apply these ideas to heterogeneous graphs derived from the MIMIC medical dataset [22, 21].

1.1 Related Works

There is a history of applying AI to healthcare. It begins in the 1970s with MYCIN [6], which asked a sequence of yes/no questions to physicians, to aid in identifying bacterial infections. MYCIN's inference rules were written by experts, and fixed - there was no "learning" done on new data. Modern models train on huge amounts of data in a "supervised" environment, which means that there is a target label supplied along with a datum, but human experts do not explicitly guide the model.

[9] doesn't attempt Deep Learning, and instead is concerned with extracting summary statistics from the MIMIC dataset. These include mortality by race, the most prevalent diseases, and 90-day death-after-discharge. Although no ML is done in this paper, it makes clear that EHRs contain useful information which can be analysed using computers.

[14] is an attempt to use neural networks, with the objective of predicting LoS. They use a fully connected neural network, and deal with the variable input size by randomly sampling patient data up to some fixed number. The information that they feed to model is limited to procedures and diagnoses.

[19] is much more interesting. After the invention of Transformer in [38], which is now the dominant sequence-modelling architecture in ML, [10] developed BERT, which is a large-language-model (LLM), extensively trained on English sentences. [2] trains a BERT-style model on clinical notes, and [19] uses this model in order to predict 30-day readmission. This approach is sophisticated, and aligns more closely with the ideas in this dissertation than previously mentioned works, as it uses ideas from GDL - the transformer and BERT architectures better respect the structure of clinical note data than the fixed-window random sampling above.

[15] serves as an extension to [19], which leverages patient-similarity information along with embeddings derived from clinical notes, also to predict 30-day readmission. This work outperforms [19], while using the same model from [2] to do word embedding. Thus, we see that using connectivity information can improve the effectiveness of these models.

1.2 Technology

The implementation of the ideas in this dissertation are written in Python, and rely heavily on the PyTorch [33] and PyTorch Geometric [13] libraries. I, like [19, 15], use the pre-trained ClinicalBERT model from [2], but stress that deep learning on EHRs is not necessarily locked into using this model.

Chapter 2

(Geometric) Deep Learning

2.1 Neural Networks

Neural networks form a broad class of parameterised functions $\mathcal{F}_\Theta$, which on an input, apply a sequence of operations (layers) to return an output. Based on how correct we deem this output, the network updates $\boldsymbol{\Theta}$ to (hopefully) return a better output on subsequent inputs. I cover necessary ML basics - see [30, 16] for more.

2.1.1 Fully-Connected Neural Network

A fully-connected neural network (a.k.a. multilayer perceptron / MLP) is a function composed of repeated affine transformations, interleaved with non-linear functions. A neural network can be pictured as layers of nodes, with each layer fully connected with the layers on either side. We call each of these nodes a neuron. When we apply the network to an input, we fill the first layer of neurons with the input vector, and subsequent neurons are updated as transformed weight-sums over values of each neuron in the previous layer.

Definition 2.1.1 (MLP)**.** For input dimension d_{in}, hidden dimension d_h, output dimension d_{out}, depth $L > 0$, and differentiable, element-wise activation function σ, an MLP implements the function:

$$\sigma \circ f_L \circ \sigma \circ f_{L-1} \circ \cdots \circ f_0$$

where

$$f_0 : \mathbb{R}^{d_{in}} \to \mathbb{R}^{d_h}, f_1 \cdots f_{L-1} : \mathbb{R}^{d_h} \to \mathbb{R}^{d_h}, f_L : \mathbb{R}^{d_h} \to \mathbb{R}^{d_{out}}$$

and

$$f_i(\mathbf{x}) = \mathbf{W}^i\mathbf{x} + \mathbf{b}^i, \sigma(\mathbf{x}) = (\sigma(x_1) \cdots \sigma(x_d))^T$$

with matrices and vectors being specific to each layer, i.e. the superscript i denotes the i^{th} matrix and vector, which are called the weight matrix and bias vector at layer i. More generally, we can define the above with a sequence of activation functions $\sigma_0 \cdots \sigma_L$, with σ not necessarily applied element-wise - in which case:

$$\sigma_0 \cdots \sigma_{L-1} : \mathbb{R}^{d_h} \to \mathbb{R}^{d_h}, \sigma_L : \mathbb{R}^{d_{out}} \to \mathbb{R}^{d_{out}}$$

The non-linearities σ are needed for a variety of reasons, most compellingly that without it, the network would collapse to one large affine function, which would reduce its power significantly, i.e. if $f = f_L \circ f_{L-1} \cdots \circ f_0$, with f_i defined as above, then f is implementable as $f(\mathbf{x}) = \mathbf{W}\mathbf{x} + \mathbf{b}$, with $\mathbf{W} \in \mathbb{R}^{d_{out} \times d_{in}}, \mathbf{b} \in \mathbb{R}^{d_{out}}$.

2.1.2 Optimisation

Given a parameterised model, we can try to adjust the parameters to obtain a better model. If we apply a model f to an input $\mathbf{x}$, with the hope that the network implements some other function $\tilde{f}$, it is natural to ask how well the model did at approximating $\tilde{f}$. For this, we use a **loss function**, which measures the difference between the "true" value of $\tilde{f}(\mathbf{x}) = y$, the ground truth value/label, and the result output by the network, $f(\mathbf{x}) = \hat{y}$, written as $\mathcal{L}(\hat{y}|y)$, the loss between predicted and true values. We wish for our network to minimise the loss between its predictions on data, and the ground truth values of the data. It is important to note that the labelled data that we feed the model is likely disjoint from data to which we will apply our models. This split between training and test datasets is important - our model learns to improve its performance on the training dataset, but with no guarantee that this performance will carry over to unseen, test datapoints.

The key property of a neural network that allows us to optimise its action is that it is **fully differentiable**. Using calculus, we can take the derivative of any output neuron with respect to any input neuron, intermediate neuron, or any weight / bias scalar in the network. Thus, provided that $\mathcal{L}$ is differentiable, we can optimise our network to minimise loss, using tools from continuous optimisation! In practice, we run a **forward pass** of the network on inputs $x_1, \cdots, x_N$ to obtain outputs $\hat{y}_1, \cdots, \hat{y}_N$. We then compute the loss between the outputs and true values to obtain $\mathcal{L}(y_1, \cdots, y_N|\hat{y}_1, \cdots, \hat{y}_N) = \frac{1}{N}\Sigma_{i=1}^{N}\mathcal{L}(y_i|\hat{y}_i)$, and run a **backward pass**, in which we update the parameters of the network to minimise the loss. The optimisation tool that we typically use is some form of gradient descent.

Definition 2.1.2 (Gradient Descent)**.** To minimise convex function f, gradient descent is an iterative process, taking steps in the direction of steepest descent to find a minimum. For input $\mathbf{x}_0$, one step of the process returns $\mathbf{x}_1$, from which we can run another step. Steps are of the form

$$\mathbf{x}_i = \mathbf{x}_{i-1} - \alpha_i \nabla_{\mathbf{x}} f(\mathbf{x}_{i-1})$$

where α_i is a step size, which can be fixed, or vary according to some schedule.

Although neural networks are not convex as functions of their parameters, in practice we still use gradient descent algorithms to optimise them. To use these algorithms, we need to be able to differentiate the output of a neural network with respect to its parameters, as well as differentiating the loss function with respect to the output of the network. We can calculate these iteratively, using the gradients at one layer to calculate the gradients from the previous layer. Notationally, we use $\mathbf{h}^l$ to refer to the values at layer l, after applying a linear layer and non-linear activation to $\mathbf{h}^{l-1}$. We get the equation:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}^l} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}^L} \frac{\partial \mathbf{h}^L}{\partial \boldsymbol{\theta}^l} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}^L} \frac{\partial \mathbf{h}^L}{\partial \mathbf{h}^{L-1}} \cdots \frac{\partial \mathbf{h}^l}{\partial \boldsymbol{\theta}^l}$$

where $\frac{\partial \mathbf{h}^l}{\partial \mathbf{h}^{l-1}}$ is a $d_l \times d_{l-1}$ Jacobian matrix of partial derivatives. In the case of MLP, this matrix can be decomposed as $\frac{\partial \mathbf{h}^l}{\partial \mathbf{h}^{l-1}} = \frac{\partial \mathbf{h}^l}{\partial \mathbf{a}^l} \frac{\partial \mathbf{a}^l}{\partial \mathbf{h}^{l-1}}$, where $\sigma(\mathbf{a}^l) = \mathbf{h}^l$. In this case, $\mathbf{a}$ is called the vector of "pre-activations". In the case that σ is applied elementwise (i.e. $\mathbf{h}_i = \sigma(\mathbf{a}_i)$), $\frac{\partial \mathbf{h}^l}{\partial \mathbf{a}^l}$ is diagonal, as $i \neq j \implies \frac{\partial \mathbf{h}_i^l}{\partial \mathbf{a}_j^l} = 0$.

Definition 2.1.3 (Backpropagation)**.** For neural network f, consisting of parameterised layers $f_1, \cdots, f_L$, with f_i parameterised by $\boldsymbol{\theta}_i$ such that the entire network is parameterised by $\boldsymbol{\Theta} = [\boldsymbol{\theta}_1, \cdots, \boldsymbol{\theta}_N]$; loss function $\mathcal{L}$; data $[\mathbf{x}_1, \cdots, \mathbf{x}_N]$ with associated true values $[y_1, \cdots, y_N]$; a forward pass consists of running the network on the data, to obtain $[\hat{y}_1, \cdots, \hat{y}_N]$. On the backward pass, having calculated the loss $\mathcal{L}(y_1, \cdots, y_N | \hat{y}_1, \cdots, \hat{y}_N)$ between the network's predictions $\{\hat{y}_i\}$ on $\{\mathbf{x}_i\}$ and the true values $\{y_i\}$, we take $\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}^l}$ for $\boldsymbol{\theta}^l$ at each step of the network iteratively, and then update $\boldsymbol{\Theta}$ with a step of the gradient descent algorithm to minimise $\mathcal{L}$. The process is called backpropagation, as gradient information is propagated backwards.

In practice, we calculate gradients at each step using only a random subset of available training data - this is called SGD.

2.2 GDL

Though MLP is powerful, and has the ability to learn from data, it is only the starting point for today's ML architectures. Its generalisation power is quite limited, as the architecture makes very few assumptions about the training data - in reality, the data on which we would like to run models has a lot of implicit structure. MLP is also of a fixed input / output dimensionality, which makes it very poor on datasets of varying sizes, such as graphs.

2.2.1 Curse of Dimensionality & Inductive Bias

As the data we train our models on gets complex, the dimensionality of our input grows. For example, each datum of the MNIST dataset [23], consisting of 28×28-pixel hand-drawn images of digits 0-9, is a 784 dimensional vector, with entries in $\{0,1\}$. The data have very low intrinsic dimension (estimated at 13.4 in [12]), but the number of possible 28×28 images is huge (2^{784}). An MLP trained on these images is really trained on long, flat vectors, with no notion of "image structure".

The label of an image of the number "6" does not change depending on where in the image it is found, but MLP encodes no such bias - instead we have to teach it what a 6 looks like at every possible position in the grid. This seems pathological, requires lots of training data, and means that our model likely will not generalise well. It makes more sense to "scan" the image, trying the **same** process of "looking for a number" at every position, as a human would do when trying to identify numbers. This is the approach taken in [24], with LeNet, the first convolutional neural network.

The idea from above says that there is a gap between our inductive bias as humans when identifying images, and the inductive bias that we encode into our model. As humans, we assume that the position of the digit can be ignored when classifying it. MLP has no such in-built bias.

GDL seeks to build architectures with inductive biases that respect the underlying symmetries in data, in order to break the curse of dimensionality. A symmetry here, to quote [5] is "a transformation that preserves some property or structure". Examples of these symmetries include translational and rotational symmetries in 2D image classification, 1D translational symmetry in audio classification, and 3D rotational symmetry in molecule design. Without applying geometric ideas to our models, they require far more data to train, and don't respect the structure of the data properly.

As per MLP, we still wish to build our models by composing layers that can be trained by backpropagation. Similarly, we build these structures by interleaving

different types of layers. Typically, parameterised layers are linear, or implemented using MLP, as these are easy to train. Read [5] for more on GDL.

2.2.2 Signals

Our input data is assumed to be defined on some underlying space. For images, that space is the 2D grid. A datum can be thought of as a function on the underlying domain, called a signal, which maps points in the space to some problem-specific codomain.

Definition 2.2.1. A signal is a function $x : \Omega \to \mathcal{C}$. We write $\mathcal{X}(\Omega, \mathcal{C})$ to denote the space of signals, defined on domain Ω, with values lying in $\mathcal{C}$, the **feature space**.

Remark. For the MNIST dataset $\Omega = \{1, \cdots, 28\}^2, \mathcal{C} = \{0, 1\}$.

The learned function $f_{\boldsymbol{\theta}}$ that our models implement is drawn from some parameterised class $\mathcal{F}_{\boldsymbol{\Theta}}$. We wish to use knowledge of data symmetries to constrain this function class appropriately.

2.2.3 Group Theory

The language of symmetries is group theory. I assume some familiarity with the basic properties of mathematical groups. This section makes rigorous the notion of shift-invariance in 2D images, and data symmetries generally.

Definition 2.2.2 (Group Action)**.** For a group $\mathcal{G}$, and set Ω, the action of $\mathcal{G}$ on Ω, written $g \cdot u$ for $g \in \mathcal{G}, u \in \Omega$, is a function $\cdot : \mathcal{G} \times \Omega \to \Omega$, such that $g \cdot (h \cdot u) = (gh) \cdot u$, and that for the identity e of the group $\mathcal{G}$, $e \cdot u = u$ for all u.

We omit the $\cdot$ in $g \cdot u$ henceforth (unless unclear in context).

Definition 2.2.3 (Group action on signals)**.** For a group $\mathcal{G}$ acting on Ω, we derive an action of $\mathcal{G}$ on $\mathcal{X}(\Omega, \mathcal{C})$. The result of an element $g \in \mathcal{G}$ acting on a signal $x \in \mathcal{X}(\Omega, \mathcal{C})$ is another signal, defined as $gx = \tilde{x}$, where $\tilde{x}(u) = x(g^{-1}u)$, for $u \in \Omega$.

Remark. The choice of symmetry group is dependent not only on the domain Ω, but also on the specific task.

Having chosen the group and domain correctly, we want our models to be constant along each orbit of the group, when acting on signals. For example, for the group of translations of the grid, we want our model to implement shift-invariant labelling. A more complex architecture for computer vision could consider signals acted on by

rigid motions (translations and rotations), as in [7]. In this case, we would want the label to be unchanged after applying translations and rotations. This is obviously too broad for digit classification, as the label of an image of the number 6 is altered when rotated 180°.[1]

The following definitions formalise how our architectures ought to "respect" data symmetries.

Definition 2.2.4 (Invariance)**.** For a function $\mathbf{F} : \Omega_1 \to \Omega_2$, with a group $\mathcal{G}$ acting on Ω_1, we say that $\mathbf{F}$ is $\mathcal{G}-$**invariant** if

$$\forall g \in \mathcal{G}, \forall u \in \Omega_1, \mathbf{F}(gu) = \mathbf{F}(u)$$

Remark. We can extend $\mathcal{G}$-invariant functions on Ω to $\mathcal{G}$-invariant functions defined on signals over Ω. $\mathbf{F} : \mathcal{X}(\Omega_1, \mathcal{C}_1) \to \mathcal{X}(\Omega_2, \mathcal{C}_2)$ is $\mathcal{G}$-invariant if $\forall g \in \mathcal{G}$, $\forall x \ \in \mathcal{X}(\Omega, \mathcal{C})$: $\mathbf{F}(gx) = \mathbf{F}(x)$.

That is, a function is $\mathcal{G}$-invariant if it is unchanged by the action of $\mathcal{G}$ on its input. Our hypothesis class of functions $\mathcal{F}$ will ideally be $\mathcal{G}$-invariant (and therefore constant on orbits of $\mathcal{G}$ acting on $\mathcal{X}(\Omega, \mathcal{C})$). However, if we want to build complex GDL architectures, we cannot collapse all symmetry information immediately - we should use layers that are $\mathcal{G}$-invariant, but not early in the model. For example, if we are building a computer vision architecture for identifying semantic relationships between items in an image, we **shouldn't immediately** apply a translation invariant layer, as the relative position of objects will be lost. Intuitively, we ought to do transformations of the data which respect the original positions in the image, and then apply an invariant operation at the end. This motivates the following definition.

Definition 2.2.5 (Equivariance)**.** For a function $\mathbf{F} : \Omega_1 \to \Omega_2$, with a group $\mathcal{G}$ acting on both Ω_1 and Ω_2, we say that $\mathbf{F}$ is $\mathcal{G}-$**equivariant** if

$$\forall g \in \mathcal{G}, \forall u \in \Omega, \mathbf{F}(g \cdot_1 u) = g \cdot_2 \mathbf{F}(u)$$

where $(\cdot_i)$ denotes the group action of $\mathcal{G}$ on Ω_i.

Remark. We also extend $\mathcal{G}$-equivariant functions on Ω to $\mathcal{G}$-equivariant functions defined on signals on Ω. Here, $\mathbf{F} : \mathcal{X}(\Omega_1, \mathcal{C}_1) \to \mathcal{X}(\Omega_2, \mathcal{C}_2)$ is $\mathcal{G}$-equivariant if $\forall g \in \mathcal{G}$, $\forall x \ \in \mathcal{X}(\Omega, \mathcal{C}) : \mathbf{F}(g \cdot_1 x) = g \cdot_2 \mathbf{F}(x)$

[1] This example informs the intuition that a larger symmetry group restricts a model's discriminating power by doing more weight-sharing (similar to reducing its VC dimension), but improves its inductive bias, training, and generalisation.

Informally, a function is $\mathcal{G}$-equivariant if it commutes with the group action of $\mathcal{G}$. This means that the a symmetry in the data is somehow preserved after the action of an equivariant layer.

2.2.4 Non-linearities

Similarly to the need to interleave non-linear layers in the MLP to make the model more expressive, there is a similar need to interleave non-linearities between equivariant functions. I won't elaborate further on this, and again delegate to [5].

2.2.5 Scale Separation

In image analysis, although the input data are fine-grained, it is often macroscopic information that informs an input's label. In digit classification, we could feed 256×256 images into a model, but its performance would likely be similar if we first digitally compressed these to 32×32 resolution. In the language of signals, the 32×32 grid is a **coarsening** of the 256×256 grid. Informally, a coarsening aggregates together "nearby" points to form a "lower resolution" domain. On images, for a coarse graining operator P lowering image resolution, $P : \mathcal{X}(\Omega, \mathcal{C}) \to \mathcal{X}(\tilde{\Omega}, \mathcal{C})$, where $\Omega = \{1, \cdots, 256\}^2, \tilde{\Omega} = \{1, \cdots, 32\}^2$.

The notion of "closeness" (formally, a metric on Ω) can also be used to apply functions locally, rather than globally. This will be important on large graphs, where it is computationally intractable to have all data in the input domain interacting with one another. We can do this by applying a $\mathcal{G}$-invariant function to the **neighbourhood** of each $u \in \Omega$ to obtain a globally $\mathcal{G}$-equivariant function, which doesn't rely on long-range interactions. This metric is task and Ω specific, but should be $\mathcal{G}$-invariant.

Chapter 3

Deep Learning on Graphs

I assume some knowledge of graphs. Informally, a graph is defined as V, a set of objects (nodes), and $E \subseteq V \times V$ a set of links between pairs of objects (edges). They are a very natural way of structuring information, with entities, and connections/interactions between them. For example:

- Social media, with users (nodes) and interactions (edges).
- Traffic prediction, with roads (edges) and intersections (nodes)
- Drug discovery, with atoms (nodes) and bonds (edges)

3.0.1 Problem Space

The problem space in graph ML is quite broad - we have tasks at graph-level, such as solubility prediction of molecules [32], node level, such as classification of spam on the internet [4], and edge level, such as link prediction and recommendation [40]. To unify these problems, in graph ML, we typically use models to derive node embeddings, and then use these embeddings for downstream tasks. On node classification, we can apply MLP to final node embeddings. On link prediction between u, v, we can apply MLP to their concatenated feature vectors, or calculate a similarity metric, e.g. cosine similarity. On graph classification, we can aggregate the embeddings and again apply MLP. That we derive representations for nodes motivates the name Graph Representation Learning (GRL).

3.1 GDL on graphs

To derive GNN architecture, we first consider the shape of the data, and what symmetry group acts on it. I assume in this section that a graph is undirected, without

edge features - however the architecture presented is easily extended to incorporate both of these traits.

The inputs to our models are graphs of the form $G = (V, E, \mathbf{X})$, with V, E as above, and $\mathbf{X} \in \mathbb{R}^{|V| \times d}$ the **node feature matrix**, containing transposed feature vectors $\mathbf{x}_v$ for each $v \in V$, for embedding dimensions d. If the nodes of V aren't assumed to lie in $\{1, \cdots, n\}$, $\mathbf{X}$ can instead be thought of as a function $\mathbf{X} : V \to \mathbb{R}^{d_1}$. In the spirit of the GDL framework, we work first with graphs without features, and treat $\mathbb{R}^d$ as a feature space as defined previously.

I will use the notation $G = (V, E)$ to denote a graph G, with node and edge sets as above. I use the notation $\mathbb{G}_n$ to denote the set of all graphs over the vertices $1, \cdots, n$, i.e. $V = \{1, \cdots, n\}, E \subseteq V \times V$, and $\mathbb{G} = \bigcup_n \mathbb{G}_n$.

Definition 3.1.1 (Isomorphism of Graphs)**.** We say that two graphs $G_1 = (V, E_1)$, $G_2 = (V, E_2)$, with $|V| = N$, $V \cong \{1, \cdots, N\}$, are isomorphic ($G \cong G'$) if there exists a permutation $\sigma \in \Sigma_n$ such that:

$$\forall v_1, v_2 \in V, (v_1, v_2) \in E_1 \iff (\sigma(v_1), \sigma(v_2)) \in E_2$$

That is, two graphs over the same node set are isomorphic if there exists an edge-preserving automorphism on their node set.

The symmetry group which acts on graph-structured data is the permutation group Σ_n. One natural reason for this, as seen above, is that a graph is (roughly) a set with minimal additional structure, namely connectivity. Therefore it is quite simple to lift notions of isomorphisms of sets to isomorphisms of graphs. Furthermore, as we have defined two graphs to be isomorphic if there exists a permutation between them, it is natural that the group under which we desire invariance is the symmetry group.

Definition 3.1.2 (Action of permutation group on a graph)**.** The action $\cdot_\Sigma$ of a permutation $\sigma \in \Sigma_n$ on a graph $G = (V, E)$ is defined as follows:

$$\cdot_\Sigma : \Sigma_n \times \mathbb{G}_n \to \mathbb{G}_n$$

with:

$$\sigma \cdot_\Sigma (V, E) = (V, \tilde{E}) : \tilde{E} = \{(\sigma(u), \sigma(v)) | (u, v) \in E\} \tag{3.1}$$

On graph structured data, it is ambiguous whether the graph structure is part of the underlying space Ω, or a feature of the data. For example, when learning on molecules, we are interested in learning the connectivity between nodes. However,

in the medical tasks presented here, we care less about learning structures and more about propagating information around the network. Therefore, we treat input graphs as signals, with the underlying graph structure as Ω, and node features as $\mathcal{C}$.

Remark. With the above definition in mind, that $\Omega = \mathbb{G}, \mathcal{C} = \mathbb{R}^d$, we obtain an action of Σ on $\mathcal{X}(\Omega, \mathcal{C})$ as previously. Note that σG is only defined for $\sigma \in \Sigma_n, G \in \mathbb{G}_n$ for **the same n**. In terms of feature matrices, if $\mathbf{X}$ is the feature matrix for graph G, then the feature matrix $\tilde{\mathbf{X}}$ for σG is defined as $\tilde{\mathbf{X}}_i = \mathbf{X}_{\sigma^{-1}(i)}$.

Theorem 3.1.1. *For all $G = (V, E) \in \mathbb{G}_n$, the action of any permutation $\sigma \in \Sigma_n$ maps G to an isomorphic graph. That is, $\forall G \in \mathbb{G}_n, \forall \sigma \in \Sigma_n, \sigma G \cong G$*

Proof. Trivial. Follows from 3.1.1. □

Remark. This makes rigorous the notion that a graph is defined "up to relabelling".

When we load a graph into computer memory, we put an arbitrary ordering on it - the indexing of the feature matrix, based on positions in memory. As seen above, we want our architecture to ultimately ignore this ordering. Σ_n-equivariant functions can be reframed as preserving this arbitrary ordering (while being agnostic to it), while Σ_n-invariant functions are not only agnostic to the ordering, but also collapse the ordering information on a forward pass. As we derive node embeddings in GRL, we cannot be completely Σ_n-invariant, as the ordering of input nodes determines the ordering of output embeddings - rather we want Σ_n-equivariant models.

3.2 GNN

We want GNN architecture comprising permutation-equivariant layers. To do this, we apply a permutation-invariant function to the neighbourhood of each node, to obtain a globally permutation-equivariant layer.

3.2.1 GNN Architecture

A GNN layer is typically implemented using some form of **message-passing**.

Definition 3.2.1. A **message passing layer** is a function $\mathbf{F}_t : \mathbb{R}^{N \times d_{t-1}} \to \mathbb{R}^{N \times d_t}$, where $t \in [1, L]$ indexes the layers of the network. Row i of the input matrix contains the transposed feature vector for node i; row i of the output matrix contains the updated representation of node i. Thus, assuming we apply multiple layers in the message-passing neural network, we get a sequence of representations $\{\mathbf{h}_i^t\}$ of each

node i, at each layer t. The action of $\mathbf{F}$, in its most general form, updates node features as follows:

$$\mathbf{h}_u^t = \mathbf{\Phi}_t(\mathbf{h}_u^{t-1}, \bigoplus_{v \in \mathcal{N}(u)}^{t} \{\{\Psi_t(\mathbf{h}_u^{t-1}, \mathbf{h}_v^{t-1})\}\})$$

where:

- $\mathcal{N}(u)$ is some "neighbourhood" of u. Adjacent nodes in the graph is typical.
- Ψ is the message function, which computes a digest of node u and each of its neighbours v.
- $\bigoplus$ is a permutation invariant aggregation function acting on the multiset of messages from neighbours of u (a multiset is a set where elements can appear more than once).
- $\mathbf{\Phi}$ updates the representation of a node u using its previous representation, and the aggregated messages from it neighbours.

Remark. For a network of L layers, we can have potentially L message, aggregation, and update functions.

In practice, however, these functions are typically of the form:

- $\Psi_t(\mathbf{h}_u, \mathbf{h}_v) = \alpha_{u,v}^t(u, v)\mathbf{h}_v$, where α is a attentional coefficient, encoding the importance of node v to node u based on their representations.
- $\bigoplus$ is typically sum, max, or mean.
- $\mathbf{\Phi}_t(\mathbf{h}, \mathbf{z}) = f_\Theta(\mathbf{W}_t^{self}\mathbf{h} + \mathbf{W}^{neigh}\mathbf{z})$, for $\mathbf{z}$ the aggregated neighbour features, $\mathbf{W}$ weight matrices, and f_Θ a non-linearity, either element-wise or MLP.

In the above, $\alpha, f_\Theta, \mathbf{W}$ are learnable, and distinct for each layer.

The update step of a GCN layer, a particularly simple GNN model, can succinctly be written as $\mathbf{X}^{t+1} = \sigma(\tilde{A}\mathbf{X}^t\mathbf{W}^t)$, where $\tilde{A}$ is the normalised adjacency matrix (with self-loops), encoding connections between nodes, and controlling the diffusion of node features along edges, and σ is some element-wise non-linearity.

3.3 Provable power

3.3.1 Terminology

Definition 3.3.1. A **graph invariant** is a function ξ defined on graphs, such that for all $G \cong G'$:

$$\xi(G) = \xi(G')$$

Such ξ is by definition permutation-invariant.

Definition 3.3.2. A **node invariant** is a function ξ defined on graph-node pairs, such that for all graphs G, all automorphisms f of G, and all nodes $v \in V_G$:

$$\xi(G, v) = \xi(f(G), f(v))$$

Such ξ is by definition permutation-equivariant.

Definition 3.3.3. A graph invariant ξ is **sound** if $\xi(G) \neq \xi(G') \implies G \ncong G'$. A node invariant ξ is **sound** if $\xi(G, v) \neq \xi(G', v')$ implies that there is no isomorphism mapping G to G', taking v to v'. In these cases, we say that ξ **distinguishes** G and G', or (G, v) and (G', v') respectively, written $G \mid_\xi G'$ if so, and $G \nmid_\xi G'$ if not.

Definition 3.3.4. A graph invariant ξ is **complete** if $\xi(G) = \xi(G') \implies G \cong G'$. A node invariant ξ is **complete** if $\xi(G, v) = \xi(G', v')$ implies that there exists an isomorphism mapping G to G', taking v to v'.

Remark. Invariants are automatically sound, by the invariance / equivariance properties, but are not necessarily complete.

Definition 3.3.5 (Refinement)**.** For invariants ξ, ξ', we say that ξ **refines** ξ', written $\xi \preceq \xi'$ if $\xi(w) = \xi(z) \implies \xi'(w) = \xi'(z)$ (where w, z are either both graphs, or both graph-node pairs). That is, ξ at least as powerful a discriminator as ξ'. We say that invariants ξ, ξ' are equivalent, $\xi = \xi'$, if $\xi \preceq \xi' \wedge \xi' \preceq \xi$. [1].

Definition 3.3.6. For node invariant ξ, we can derive an associated graph invariant $\bar{\xi}(G) = f(\{\{\xi(G, v) | v \in V_G\}\})$, for some multiset-injective f. In the case of 1-WL, f is the identity. In the case of GNN, f is some "readout function".

[1] We can think of $\preceq$ as a partial order on invariants (ordering properties are all trivially satisfied)

Lemma 3.3.1. $\xi \preceq \xi' \implies \bar{\xi} \preceq \bar{\xi}'$

Proof.

$$G \not\upharpoonright_{\bar{\xi}} G' \iff \bar{\xi}(G) = \bar{\xi}(G')$$

By injectivity:

$$\{\{\xi(G, v) | v \in V_G\}\} = \{\{\xi(G', v') | v' \in V_{G'}\}\}$$

By $\xi \preceq \xi'$:

$$\{\{\xi'(G, v) | v \in V_G\}\} = \{\{\xi'(G', v') | v \in V_{G'}\}\}$$

Therefore:

$$G \not\upharpoonright_{\bar{\xi}'} G'$$

□

3.3.2 Colour Refinement

Algorithm 1 1-WL / Colour refinement

Input: $G = (V, E, \mathbf{X}_V)$
  $c_v^0 \leftarrow hash(\mathbf{X}_v)$ for $v \in V$
  while $c^t \neq c^{t-1}$ **do**
    $c_v^{t+1} \leftarrow hash(c_v^t, \{\{c_u^t | u \in \mathcal{N}(v)\}\}), \forall v \in V$
  end while
  return $(c_v^t)_{v \in V}$

The forward pass of a GNN model closely resembles the updates steps of the 1-Weisfeiler-Lehman (henceforth 1-WL) algorithm, also known as colour refinement - see [20] for details. Informally, we colour each node of the graph at initialisation, and iteratively update each node's colour based on the multiset of neighbouring colours at the previous step (this update function is injective).[2] We terminate when successive colourings are equivalent. 1-WL actually belongs to a family of k-WL algorithms (which have strictly increasing representational power), which can be thought of as colouring k-tuples of nodes.

Definition 3.3.7 (Equivalent colourings)**.** Two colourings $(c_v^i)_{v \in V}, (c_v^j)_{v \in V}$ of the nodes $v \in V$ are equivalent if $c_{v_1}^i = c_{v_2}^i \iff c_{v_1}^j = c_{v_2}^j \forall v_1, v_2 \in V$. That is, the colourings have equivalent discriminatory power on V. With an abuse of notation, I write $c^i = c^j$ if the colourings c^i, c^j on V are equivalent (V obvious in context).

[2] Each colouring refines the previous colouring (and therefore all previous colourings, trivially, by transitivity).

Colour refinement generates a sequence of node invariants, with each successive colouring refining the previous, until two successive invariants are equivalent. That each colouring refines the previous is trivial.

The importance of 1-WL is two-fold. Firstly, the 1-WL isomorphism test can be done by running the colour refinement algorithm on two graphs simultaneously. If the graphs are isomorphic, then the multisets of colours reached at convergence will be the same. Thus, the 1-WL test is a necessary but insufficient test for graph isomorphism - it is a sound, but incomplete graph invariant. Secondly, the power of an L-layer GNN to distinguish nodes is upper-bounded by that of the 1-WL algorithm run for L steps [39].

In fact, 1-WL can be **too** discriminatory. On two similar but non-identical inputs, 1-WL may return different colourings, even though we want similar output embeddings. We can re-frame GNN as learning a "less injective" version of 1-WL, so that similar graphs / nodes end up with similar labels.

Chapter 4

Data

The MIMIC dataset [22, 21] is a large medical dataset, consisting of EHRs collected from a hospital in Boston. The dataset is a relational database, containing anonymised records of patients' stays at hospital, including prescriptions, diagnoses, procedures and clinical notes.

4.1 Data format

Documentation is available at [1]. The database, roughly speaking, contains three types of table: definitions, events, and codes.

- The definition tables define the patients, their admissions, and links between them; and assign unique admission and patient IDs, which can be used to track information on a particular patient across different tables.
- The codes tables define mappings from human-readable names of drugs, procedures, and diagnoses to short numerical identifiers.
- The events tables track interactions between patients, identified by an admission ID, and items in the codes tables, identified with their short code, along with additional, item-specific information, such as dosage, date, duration.

4.1.1 Discharge Summary

A discharge summary is a piece of text, written by a doctor, which records information about a patient's stay in hospital. This includes the original reason for admission, test results, and drugs prescribed, procedures done, or diagnoses made during their stay. Importantly, all of this information is recorded in the MIMIC tables - a doctor

summarises it in human readable English after a patient leaves the hospital, to inform the patient's GP and improve their future care.

4.1.2 Deriving a graph

We can translate this relational database to a graph quite easily, with a nodeset taken from the rows of the definitions and codes tables as nodes, and an edge set from the events tables. A small subsection of this graph is seen in 4.2. There are many more tables in the dataset than I use - I restrict this modelling to prescriptions, diagnoses, and procedures. Other works have used physiological time-series [17] and radiology reports [37].

The seemingly easy creation of this graph comes with a couple of caveats. Firstly, the graph is large: in total, this graph has well over 120000 nodes (though most of these are individual patient or visit nodes), and over 4,000,000 edges (most of these are drug prescriptions). The data object is seen in 4.1. Secondly, and more interestingly, it is heterogeneous. That is, there are a variety of node and edge types. I explore this in the next chapter.

4.1.3 Derived features

The nodes of this graph need informative features in order to do learning. I use ClinicalBERT [2] and Sentence-BERT [35] to embed the human readable names for drugs, diagnoses, and procedures into $\mathbb{R}^{768}$ (with the assumption that these node features are unique). This work is not really concerned with *how* we derive features, but only assumes that they are informative - other works have done embeddings using ICD-9 code [26]. The admission nodes are initialised with uninformative features.

4.2 Problem definition

I wish to test whether a graph-based approach can successfully predict readmission of patients. Other works, such as [15], have used a neighbourhood graph embedding as the second step in a two-stage architecture, but use domain-specific knowledge to do the original encoding. Specifically, they derive a node embedding from discharge summaries, which have already been processed by human doctors to contain all relevant information about a stay. I will try to predict readmission using **only** a graph based approach, in the hopes that the model can learn for itself over which data to attend, and what to ignore.

```
data
```

```
HeteroData(
  procs={ x=[3998, 768] },
  diags={ x=[14567, 768] },
  pres={ x=[4525, 768] },
  admission={
    num_nodes=58976,
    y=[58976, 2]
  },
  patients={ num_nodes=46520 },
  (admission, prescribed, pres)={ edge_index=[2, 4156450] },
  (admission, diagnosed, diags)={ edge_index=[2, 634709] },
  (admission, had_procedure, procs)={ edge_index=[2, 496493] },
  (admission, is_patient, patients)={ edge_index=[2, 58976] }
)
```

Figure 4.1: HeteroData object containing the nodes and edges from MIMIC tables.

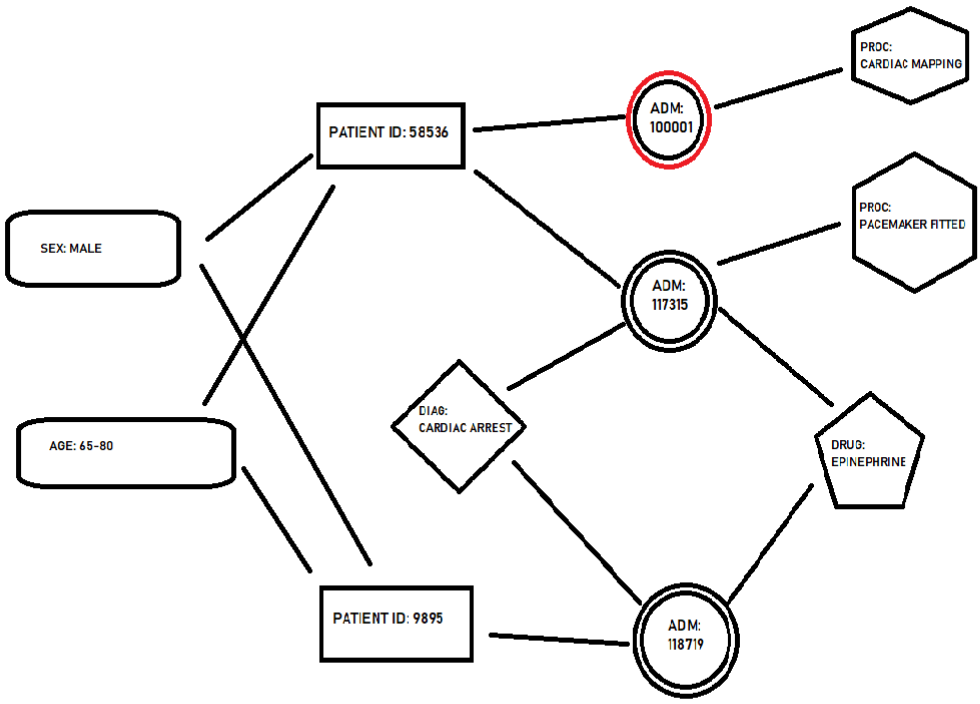

Figure 4.2: A subsection of the MIMIC graph. Node type is indicated by shape, and positive / negative readmission labels by red / black border.

Chapter 5

Heterogeneous Graph Learning

5.1 Heterogeneous Graphs

A heterogeneous graph (a.k.a. Heterogeneous Information Network, or "HIN") can be thought of as a graph with additional information structure, in the form of type labels for edges and nodes. As G refers to a graph, here H refers to a HIN.

Definition 5.1.1. A **heterogeneous graph** is defined by the tuple $H = (V, E, s, t, \mathbf{X}, \mathbf{Z}, \mathcal{A}, \mathcal{R}, \phi, \psi)$, where:

- V is a set of nodes. Unlike homogeneous graphs, these can be of differing types. In the medical domain, some may be patients, some may be drugs.
- E is a set of edges. Unlike homogeneous graphs, there can be multiple edges between two nodes, but with the convention that there is a maximum of one edge of each type connecting two nodes.
- s is the source function, $s : E \to V$. For an edge $e \in E$, $s(e) = u$, where u is the source of the directed edge e.
- t is the target function, $t : E \to V$. For an edge $e \in E$, $s(e) = v$, where v is the target of the directed edge e.
- $\mathcal{A}$ is the set of node types (objects).
- $\mathcal{R}$ is the set of edge types (relations).
- $\mathbf{X}$ is the node feature map. It is not necessarily the case that nodes of differing types contain features in the same space, so $\mathbf{X} : V \to \bigcup_{a \in \mathcal{A}} \mathbb{R}^{d_a}$.
- $\mathbf{Z}$ is the edge feature map. Similarly, $\mathbf{Z} : E \to \bigcup_{r \in \mathcal{R}} \mathbb{R}^{d_r}$.

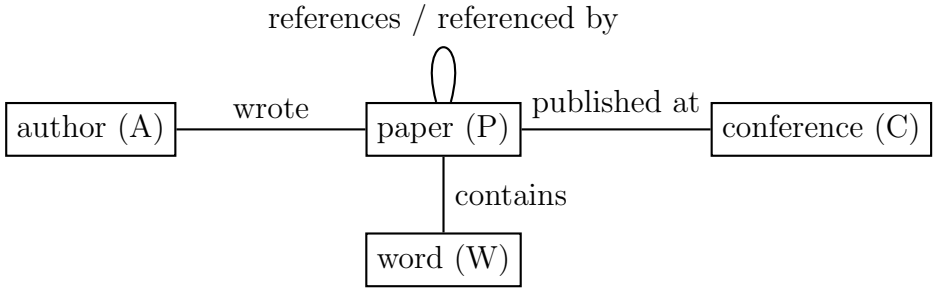

Figure 5.1: Data schema for an academic network. Here, the network has been made undirected.

- $\phi : V \to \mathcal{A}$ is the node type function, returning node types.
- $\psi : E \to \mathcal{R}$ is the edge type function, returning edge types.

For an edge $e \in E$, we call $\phi(s(e))$ the **object type** and $\phi(t(e))$ the **target type**. These node types are implicit in a relation type,

i.e. $\forall r \in \mathcal{R} \exists a_1, a_2 \in \mathcal{A} : \forall e : \psi(e) = r,\ \phi(s(e)) = a_1 \wedge \phi(t(e)) = a_2$.

If any two objects are connected by at most one relation, I call the HIN **unambiguous**.

Remark. For $|\mathcal{R}| = 1, |\mathcal{A}| = 1$, we recover a typical directed (homogeneous) graph.

Remark. a_1 and a_2 as above need not be distinct - there can be edges between nodes of the same type.

Remark. To make the HIN undirected, we implicitly fill out E with backward edges, and $\mathcal{R}$ and $\mathbf{Z}$ with reversed edge types and features (which are task specific - can be uninformative, or copied from forward edges).

Definition 5.1.2 (Network Schema)**.** The associated schema for a heterogeneous graph is a template for information structure. Given $\mathcal{A}, \mathcal{R}$, it is a directed multigraph, with $V = \mathcal{A}$, and $E \cong \mathcal{R}$, which shows the connectivity between node and edge types.

5.1.1 Example

The canonical HIN example is an academic network, for which a schema is shown in 5.1. The feature spaces for the different node types are distinct: an author has a name, an age, and other human information; a paper has a title, topic, and publishing date; a conference has a date, a venue, and an academic field.

The network has been made undirected, and is ambiguous, due to multiple relations between papers.

It is unclear immediately how to infer non-trivial information from a HIN. In the case of the academic network, the relationship between two authors isn't necessarily obvious - there aren't direct edges between them. And yet, it is intuitively clear that authors who co-publish a paper are "close". The **meta-path** is the object which makes this proximity rigorous.

Definition 5.1.3 (Meta-path)**.** A meta-path p is a path on the network schema, and is of the form $A_1 \xrightarrow{R_1} A_2 \xrightarrow{R_2} \dots \xrightarrow{R_l} A_{l+1}$, which describes a composite relation between objects A_1, A_{l+1}. For unambiguous HIN, we write this succinctly as $p = A_1 A_2 \dots A_{l+1}$. If the meta-path is symmetrical, i.e. $R_i = R_{l+1-i}$, we call the meta-path **symmetric**.

We see from this definition that there isn't a unique, canonical idea proximity in an HIN, but it is instead relation dependent. This contrasts homogeneous graphs, where proximity is understood in terms of n-hop neighbourhoods. In the academic network, we can consider the meta-paths $p_1 = APA$, $p_2 = APWPA$, and $p_3 = APCPA$. From each meta-path we can derive different measures of similarity between authors:

- p_1 encodes co-authoring of papers, suggesting relationships like those between colleagues, or between students and teachers.
- p_2 encodes similarity of publications, suggesting connected authors are within the same academic niche.
- p_3-connected authors have published papers at the same conference(s) - this may indicate that they are similarly prestigious.

5.2 Problems with Naive GRL on HIN

It would be easy now to think that we can directly apply GNN architecture to HINs. The nodes, edges, and features resemble a homogeneous graph, but heterogeneity complicates things:

- Node features reside in differing latent spaces. We can't naively use GNN architecture out of the box, as messages may be the wrong shape.
- Locality in HIN is not the same as in a homogeneous graph (meta-path vs edges) - with GNN we cannot directly send messages between nodes that don't have a relation between them, e.g. on MIMIC HIN, no edges connecting admissions.

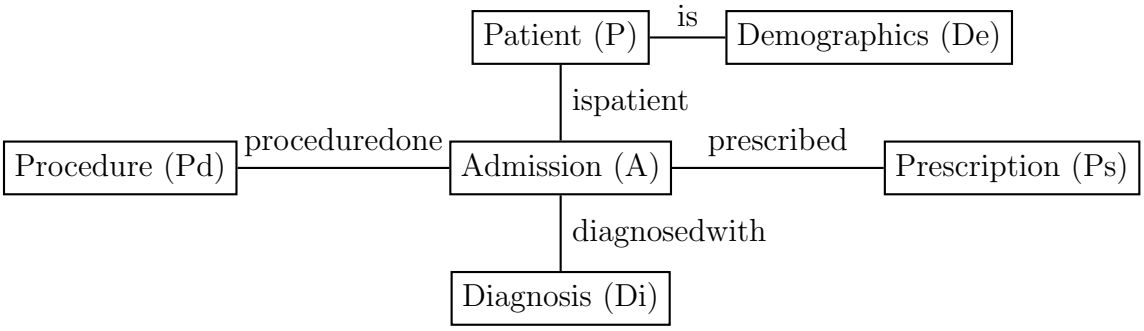

Figure 5.2: Data schema for the MIMIC HIN.

- The existence of hub nodes can lead to oversmoothing. In the case of MIMIC, demographic nodes, or popular prescriptions, are connected to a majority of patients. The propagation of these hub nodes' features can cause the rest of the nodes in the network to look very similar after a few message-passing layers.

5.3 GDL on HIN

5.3.1 Simplifying assumptions

The analysis that follows makes a few simplifying assumptions. These are:

- HIN is unambiguous.
- No edge features.
- HIN is undirected.

The first assumption reduces the ability to model multiple relations between two objects, e.g. prescriptions of, and allergies to drugs. We can discard s, t, as these are required only to deal with edge multiplicity. Thus, $E \subseteq V \times V$. The second assumption limits our ability to model fine detail - we might register a prescription as an edge from a visit node to drug node, with edge label registering the amount / route; we still model a prescription as an edge, but include no edge features. Thus, message passing will need to take edge-type into account, but not read in or update edge features. Without these assumptions the analysis gets more complex, and I leave further investigation to a later work - the MIMIC HIN is unambiguous regardless.

5.3.2 HIN Isomorphism

Definition 5.3.1 (Isomorphic HINs)**.** We say that HINs $H_1 = (V, E_1, \phi_1, \psi_1)$, $H_2 = (V, E_2, \phi_2, \psi_2)$, $|V| = N, V \cong \{1 \cdots N\}$, with node and edge types drawn from the same sets $\mathcal{A}, \mathcal{R}$, are isomorphic if there exists a **permutation** $\sigma \in \Sigma_N$ such that the following hold:

1. $\forall v_1, v_2 \in V : (v_1, v_2) \in E_1 \iff (\sigma(v_1), \sigma(v_2)) \in E_2$
2. $\forall v \in V : \phi_1(v) = a \in \mathcal{A} \iff \phi_2(\sigma(v)) = a \in \mathcal{A}$
3. $\forall (v_1, v_2) \in E_1 : \psi_1((v_1, v_2)) = r_1 \in \mathcal{R} \iff \psi_2((\sigma(v_1), \sigma(v_2))) = r_1 \in \mathcal{R}$

Informally, two HINs are isomorphic if there exists an edge-preserving permutation on their nodeset, which also preserves types.

Similar to homogeneous graphs, we treat $\mathbf{X}$ as a feature, not as a part of the underlying structure. **However**, the node and edge types are considered to be part of the domain. That is, for $\mathbb{H}_{\mathcal{A},\mathcal{R}}$, the set of all HINs over types $\mathcal{A}, \mathcal{R}$, $\Omega = \mathbb{H}_{\mathcal{A},\mathcal{R}}, \mathcal{C} = \bigcup_{a \in \mathcal{A}} \mathbb{R}^{d_a}$, and Ω is acted upon by Σ. However, our **local** $\mathcal{G}$-invariant function is not invariant to the whole permutation group Σ, but to the subset that preserves type information.

5.4 Heterogeneous message-passing

With the above in mind, we can derive an HGNN layer by applying a local function to the neighbourhood of each node, which (unlike GNN) can treat neighbours differently, based on edge types:

$$\mathbf{h}_u^t = \Phi(\mathbf{h}_u^{t-1}, \bigoplus_{v \in \mathcal{N}_u} \{\{\Psi(\mathbf{h}_u^{t-1}, \mathbf{h}_v^{t-1}, \psi((u, v)))\}\}, \phi(u))$$

where Ψ, Φ take relevant types as input.

5.4.1 Expressive Power

This layer *looks* more complex than GNN, but in the unambiguous case, it is actually no more powerful than the base GNN model! This proof requires the assumption that the sets of initial node features of distinct node types are disjoint (true of the MIMIC HIN), and that these disjoint representational subspaces are preserved throughout the model. These assumptions are quite natural - embedding nodes of differing types the same way seems pathological - but analysis of 1-WL under weaker assumptions

is out of scope. I use c_u^t, $\hat{c}_u^t$ and h_u^t to denote 1-WL & 1-HWL (to follow) colourings, and HGNN embeddings respectively, of node u after t layers / iterations.

With ζ^L referring to L steps / layers of ζ:

Theorem 5.4.1. *1-WL$^L \preceq$ HGNN^L.*

Proof. The proof follows via induction, similarly to [39]. At initialisation, 1-WL cannot distinguish nodes $u, v \iff$ HGNN cannot (hash injectivity). Assume 1-WL$^k \preceq$ HGNN^k. For iteration $k+1$:

$$c_u^{k+1} = c_v^{k+1} \implies (c_u^k, \{\{c_w^k | w \in \mathcal{N}(u)\}\}) = (c_v^k, \{\{c_w^k | w \in \mathcal{N}(v)\}\})$$

$\implies$ (induction):

$$(h_u^k, \{\{h_w^k | w \in \mathcal{N}(u)\}\}) = (h_v^k, \{\{h_w^k | w \in \mathcal{N}(v)\}\})$$

$\implies$ (disjoint subspaces assumption):

$$((h_u^k, \phi(u)), \{\{(h_w^k, \phi(w)) | w \in \mathcal{N}(u)\}\}) = ((h_v^k, \phi(v)), \{\{(h_w^k, \phi(w)) | w \in \mathcal{N}(v)\}\})$$

$\implies$ (unambiguity):

$$((h_u^k, \phi(u)), \{\{(h_w^k, \psi((u,w)) | w \in \mathcal{N}(u)\}\}) = ((h_v^k, \phi(v)), \{\{(h_w^k, \psi((v,w)) | w \in \mathcal{N}(v)\}\})$$

$\implies$ (Trivially):

$$(h_u^k, \phi(u), \{\{(h_u^k, h_w^k, \psi((u,w))) | w \in \mathcal{N}(u)\}\})$$

$$=$$

$$(h_v^k, \phi(v), \{\{(h_w^k, h_w^k, \psi((v,w))) | w \in \mathcal{N}(v)\}\})$$

$\implies$ (HGNN definition):

$$h_u^{k+1} = h_v^{k+1}$$

□

I stress that the 1-WL hash is unaware of node typing - hence the need for assumptions. To discard assumptions, I introduce the heterogeneous 1-WL algorithm (1-HWL).

Algorithm 2 1-HWL

Input: $H = (V, E, \mathbf{X}_V, \mathcal{A}, \mathcal{R}, \phi, \psi)$

$c_v^0 \leftarrow hash(\mathbf{X}_v, \phi(v))$ for $v \in V$

while $c^t \neq c^{t-1}$ **do**

$c_v^{t+1} \leftarrow hash(c_v^t, \{\{(c_u^t, \psi((v,u)))|u \in \mathcal{N}(v)\}\}, \phi(v))$, $\forall v \in V$

end while

return $(c_v^t)_{v \in V}$

Theorem 5.4.2. *1-HWL$^L \preceq$ HGNN^L.*

Proof. Similar to 5.4.1. At initialisation, 1-HWL cannot distinguish nodes $u, v \implies$ HGNN cannot (hash injectivity). Assume 1-HWL$^k \preceq$ HGNN^k.

For iteration $k+1, \hat{c}_u^{k+1} = c_v^{k+1} \implies$

$$(\hat{c}_u^k, \phi(u), \{\{(\hat{c}_w^k, \psi((u,w)))|w \in \mathcal{N}(u)\}\}) = (\hat{c}_v^k, \phi(v), \{\{(\hat{c}_w^k, \psi((v,w)))|w \in \mathcal{N}(v)\}\})$$

$\implies$ (induction):

$$(h_u^k, \phi(u), \{\{(h_w^k, \psi((u,w)))|w \in \mathcal{N}(u)\}\}) = (h_v^k, \phi(v), \{\{(h_w^k, \psi((v,w)))|w \in \mathcal{N}(v)\}\})$$

$\implies$ (Trivially):

$$(h_u^k, \phi(u), \{\{(h_u^k, h_w^k, \psi((u,w)))|w \in \mathcal{N}(u)\}\})$$

$$=$$

$$(h_v^k, \phi(v), \{\{(h_w^k, h_w^k, \psi((v,w)))|w \in \mathcal{N}(v)\}\})$$

$\implies$ (HGNN definition):

$$h_u^{k+1} = h_v^{k+1}$$

□

Lemma 5.4.3. *1-HWL terminates at $T \iff \forall u, v$:*

$$\hat{c}_u^{T-1} = \hat{c}_v^{T-1} \implies \{\{(\hat{c}_w^{T-1}, \psi((u,w))|w \in \mathcal{N}(u))\}\} = \{\{(\hat{c}_w^{T-1}, \psi((v,w))|w \in \mathcal{N}(v))\}\}$$

Proof. $\Leftarrow$ follows by 1-HWL definition. To prove $\Rightarrow$, I show the contrapositive: Assume $\exists u, v, : \hat{c}_u^{T-1} = \hat{c}_v^{T-1}$, but:

$$\{\{(\hat{c}_w^{T-1}, \psi((u,w))|w \in \mathcal{N}(u))\}\} \neq \{\{(\hat{c}_w^{T-1}, \psi((v,w))|w \in \mathcal{N}(v))\}\}$$

$\implies$

$$hash(\hat{c}_u^{T-1}, \phi(u), \{\{(\hat{c}_w^{T-1}, \psi((u,w))|w \in \mathcal{N}(u))\}\})$$

$$\neq$$

$$hash(\hat{c}_v^{T-1}, \phi(v), \{\{(\hat{c}_w^{T-1}, \psi((v,w))|w \in \mathcal{N}(v))\}\})$$

$\implies \hat{c}_u^T \neq \hat{c}_u^T$, therefore 1-HWL will not terminate at T. □

Lemma 5.4.4. $\hat{c}_u^T = \hat{c}_v^T \implies \forall t < T :$
$\{\{(\hat{c}_w^t, \psi((u,w)))|w \in \mathcal{N}(u)\}\} = \{\{(\hat{c}_w^t, \psi((v,w)))|w \in \mathcal{N}(v)\}\}$

Proof. Contrapositive states: $\exists t < T :$
$\{\{(\hat{c}_w^t, \psi((u,w)))|w \in \mathcal{N}(u)\}\} \neq \{\{(\hat{c}_w^t, \psi((v,w)))|w \in \mathcal{N}(v)\}\} \implies \hat{c}_u^T \neq \hat{c}_v^T.$
Trivially true by injectivity. □

Lemma 5.4.5. *1-HWL$^L \preceq$ 1-WL^L.*

Proof. Induction. L=0 trivial. Assume 1-HWL$^t \preceq$ 1-WL^t.

$$\hat{c}_u^{t+1} = \hat{c}_v^{t+1} \implies (\hat{c}_u^t, \phi(u), \{\{\hat{c}_w^t|w \in \mathcal{N}(u)\}\}) = (\hat{c}_v^t, \phi(v), \{\{\hat{c}_w^t|w \in \mathcal{N}(v)\}\})$$

$\implies$ (induction):

$$(c_u^t, \{\{c_w^t|w \in \mathcal{N}(u)\}\}) = (c_v^t, \{\{c_w^t|w \in \mathcal{N}(v)\}\})$$

$\implies$

$$c_u^{t+1} = c_v^{t+1}$$

□

Theorem 5.4.6. *1-HWL $\preceq$ 1-WL*

Proof. Assume 1-HWL terminated after T steps. Thus:
$\hat{c}^T = \hat{c}^{T-1} \implies$ (5.4.3):

$$\hat{c}_u^{T-1} = \hat{c}_v^{T-1} \implies \{\{(\hat{c}_w^{T-1}, \psi((u,w))|w \in \mathcal{N}(u))\}\} = \{\{(\hat{c}_w^{T-1}, \psi((v,w))|w \in \mathcal{N}(v))\}\}$$

$\implies$ (5.4.5):

$$(c_u^{T-1}, \{\{(c_w^{T-1}, \psi((u,w))|w \in \mathcal{N}(u))\}\}) = (c_v^{T-1}, \{\{(c_w^{T-1}, \psi((v,w))|w \in \mathcal{N}(v))\}\})$$

$\implies$

$$c_u^T = c_v^T$$

Thus 1-WL terminates by $L \leq T$. Again, by 5.4.5, 1-HWL$^L \preceq$ 1-WL^L, therefore 1-HWL$^T \preceq$ 1-WL^L. □

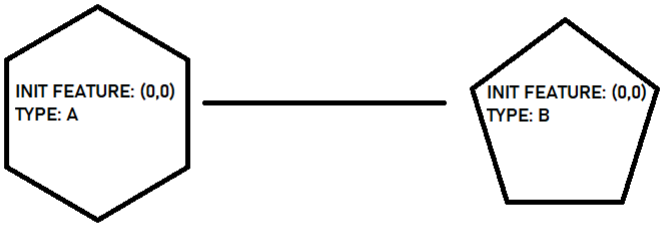

Figure 5.3: 1-WL and GNN can't distinguish these nodes: HGNN and 1-HWL can.

The example, in 5.3, and the above thoeorem, show that 1-HWL & HGNN are strictly more powerful than 1-WL & GNN. All of these proofs refer to node invariants / classification, but by 3.3.1 extend to graph invariants / classification.

We care about 1-WL's relationship to GNN as its logical power is well-understood [20]. The logical power of HWL algorithms therefore warrants similar investigation, as it is more powerful than 1-WL and provides a bound on HGNN's distinguishing power - this is sadly out of scope.

5.5 Two-stage architecture

We have considered heterogeneity, but we haven't considered the other problems of the MIMIC graph: it is large, and we cannot perform message-passing along important meta-paths (specifically between admissions). To solve both of these problems, I propose a two-step procedure (5.4):

1. We compute an embedding for each admission.
2. We use these embeddings as node features in a coarsened, homogeneous graph, with connectivity derived from meta-paths.

Here V_a denotes $v \in V : \phi(v) = a \in \mathcal{A}$, where V is the MIMIC HIN's nodeset. The architecture is deliberately modular - the node-embedding HGNN, edge embedding method, and admission graph GNN are loosely coupled.

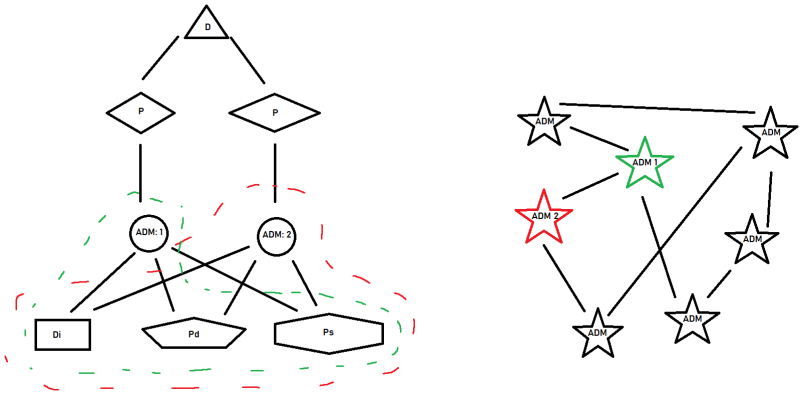

Figure 5.4: Left: section of original MIMIC graph. Right: coarsened adm graph.

Algorithm 3 HGNN-D: HGNN then Downsample

Input: MIMIC HIN H, with node types {Adm,Pat,Ps,Pd,Di,De}

$D = (V_{adm}, E_D = \{\}, \mathbf{X}_D = \mathbf{0})$ - a graph with nodeset V_{adm}, no edges or features.
run HGNN on H to obtain embeddings $\{\mathbf{h}_v | v \in V_{adm}\}$
for $v \in V_{adm}$ **do**
    $\mathbf{X}_D[v] = \mathbf{h}_v^T$, i.e. update v's feature in D.
end for
return $\mathbf{X}_D$

Algorithm 4 D-HGNN: Downsample then HGNN

Input: MIMIC HIN H, with node types {Adm,Pat,Ps,Pd,Di,De}

$D = (V_{adm}, E_D = \{\}, \mathbf{X}_D = \mathbf{0})$ - a graph with nodeset V_{adm}, no edges or features.
for $v \in V_{adm}$ **do**
    $H_v = \bigcup_{a \in \{Ps,Pd,Di\}} \{u \in V_a | u \in \mathcal{N}(v)\}$, fully connected, features from H.
    $\mathbf{h}_v = SUBGRAPH\_HGNN(H_v)$
    $\mathbf{X}_D[v] = \mathbf{h}_v^T$, i.e. update v's feature in D.
end for
return $\mathbf{X}_D$

5.5.1 Embedding HGNN (and discriminatory power)

We derive a node embedding for each admission $v \in V_a$, for use in the derived homogeneous graph D.

In the first case, we run HGNN, and then use embeddings $\mathbf{h}_v$ for $v \in V_{adm}$ as features in D. This is "HGNN-D" (HGNN-Downsample).

In the second case ("D-HGNN", Downsample-HGNN), we downsample first, to a sub-HIN (containing only prescriptions, procedures, diagnoses) H_v for each admission $v \in V_{adm}$, and then run HGNN (with some injective readout) to derive embedding $\mathbf{h}_v$ of H_v, to be used in D.

Recall that we assume uninformative initial features for patients & admissions.

I use $\mathbf{h}, \mathbf{d}$ for HGNN-D, D-HGNN embeddings respectively, $\mathbf{x}$ for initial features (clinicalBERT), and assume HGNNs are maximally powerful (i.e. injective updates and aggregation, matching 1-HWL power).

Theorem 5.5.1. *Considered as node invariants on H, V_{adm}, HGNN-D $\preceq$ D-HGNN.*

Proof. For $u, v \in V_{adm} : \mathbf{h}_u = \mathbf{h}_v \implies$ (5.4.4 + injectivity)

$$\{\{(\mathbf{x}_w, \psi((u,w)))|w \in \mathcal{N}(u)\}\} = \{\{(\mathbf{x}_w, \psi((v,w)))|w \in \mathcal{N}(v)\}\}$$

$\implies$

$$H_u = H_v \text{ (as defined in D-HGNN algorithm)}$$

$\implies$

$$\mathbf{d}_u = \mathbf{d}_v$$

□

Lemma 5.5.2. *D-HGNN $\not\preceq$ HGNN-D*

Proof. Counterexample. See 5.5.

$$\text{D-HGNN}(H, v_A) = \text{D-HGNN}(H, v_B) \text{ but } \text{HGNN-D}(H, v_A) \neq \text{HGNN-D}(H, v_B)$$

□

Theorem 5.5.3. *Considered as node invariants on $H \setminus V_{pat}, V_{adm}$, i.e. removing patient nodes (and therefore demographic information also) from the HIN, HGNN-D = D-HGNN.*

Proof. HGNN-D $\preceq$ D-HGNN as before. Remains to show D-HGNN $\preceq$ HGNN-D. For $u, v \in V_{adm}, \mathbf{d}_u = \mathbf{d}_v \implies$ (lemma 5.4.4):

$$\{\{(\mathbf{x}_w, \psi((u,w)))|w \in \mathcal{N}(u)\}\} = \{\{(\mathbf{x}_w, \psi((v,w)))|w \in \mathcal{N}(v)\}\}$$

$\implies$ (uniqueness of initial (medical) embeddings):

$$\{\{(w, \psi((u,w)))|w \in \mathcal{N}(u)\}\} = \{\{(w, \psi((v,w)))|w \in \mathcal{N}(v)\}\}$$

$\implies$ (uniformity of admission embeddings):

$$(\mathbf{h}_u^0, \phi(u), \{\{(w, \psi((u,w)))|w \in \mathcal{N}(u)\}\})$$

$$=$$

$$(\mathbf{h}_v^0, \phi(v), \{\{(w, \psi((v,w)))|w \in \mathcal{N}(v)\}\})$$

$\implies$ (that u, v have the same **neighbourhood** of nodes, not just same multiset of neighbour features, i.e. $\mathcal{N}(u) = \mathcal{N}(v)$): $\forall t$

$$(\mathbf{h}_u^t, \phi(u), \{\{(\mathbf{h}_w^t, \psi((u,w)))|w \in \mathcal{N}(u)\}\})$$

$$=$$

$$(\mathbf{h}_v^t, \phi(v), \{\{(\mathbf{h}_w^t, \psi((v,w)))|w \in \mathcal{N}(v)\}\})$$

$\implies$

$$\mathbf{h}_u = \mathbf{h}_v$$

□

We would like to use D-HGNN, as it can be parallelised, doesn't require loading the full HIN into memory, aligns better with the scale separation principle, and allows us to use more complex local architectures (e.g. fully connected models, such as Transformer). If we remove patient nodes from the HIN, the models have equivalent discriminatory power - the edge encoding therefore includes patient-based meta-paths to recover this information.

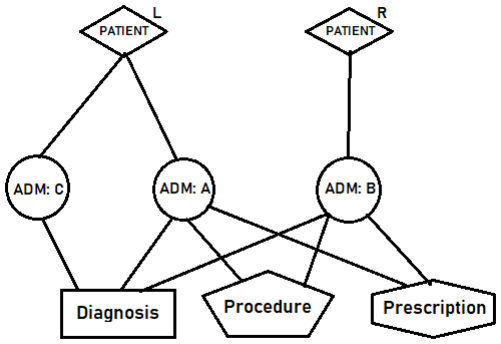

Figure 5.5: After two layers of HGNN, A and C have differing representations, as the presence of C updates patient L after 1 layer, which updates A after 2 layers.

5.5.2 Meta-path edge embedding

We wish to recover relevant meta-paths connecting admissions with our choice of edges in the coarsened graph. These include:

- Adm-Pat-Adm: same patient
- Adm-Pat-De-Pat-Adm: similar patients (e.g. age, weight, sex)
- Adm-Diag-Adm: similar diagnoses
- Adm-Pres-Adm & Adm-Proc-Adm: similar care

There are a few options for deriving edges, given meta-paths. [11] presents Meta-Path2Vec, which uses a skip-gram model (similar to [28]) to derive node embeddings in a heterogeneous network, based on meta-path connectivity (though this model is only equivariant in expectation). Using these embeddings, we can derive edges (for D) by taking cosine similarity between the learned node embeddings. Alternatively, [27] suggests "Symmetrised PathSim" (SPS), which is equivariant, and directly computes edge weights.

Chapter 6

Experiment

The goal is to test whether a GNN approach can predict readmission of patients using MIMIC table data, at a comparable level to methods using discharge summaries. Although I highlighted the full power of HGNN earlier, the implementation here uses homogeneous architecture, as full heterogeneous architecture is too complex for this dissertation. I ran this task using D-HGNN, with Transformer [38] and GIN [39], a maximally powerful GNN, as the local GNNs. Transformer has been seen to learn complex semantic relationships in natural language data in the medical domain and achieved good results in [19], which justifies this choice. I also proved that 1-WL upper bounds the expressive power of unambiguous HGNN, which motivates the use of GIN architecture.

The label distribution is skewed - of 58976 admissions, only 3227 lead to readmission. To improve training, each training epoch iterates over all positive training data, and 3227 randomly sampled negative examples. The model (with a high probability) eventually sees most of the negative training data, due to the random sampling.

The derived subgraphs are fully connected, with nodes for all diagnoses, prescriptions, and procedures done. Modelling more complex features, specifically temporality or medical knowledge, is too complex for this work - [34] does well with both of these factors included. We can view this architecture (without neighbourhood graph) as a version of [19], where we task the model first with inferring what EHR information is useful (rather than relying on the clinician), and then learning from that information; with the neighbourhood graph, it is similar to [15], with the same inference subtask.

6.1 Results

Sadly, the results are in the negative. That is, a GNN-only approach, with data comprising unstructured diagnoses, procedures, and prescriptions, is not yet sufficient to

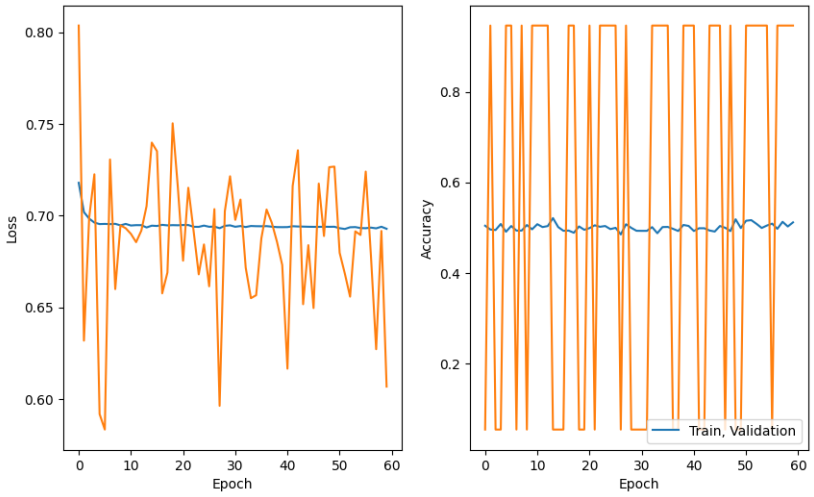

Figure 6.1: Loss and Accuracy on downsampled graphs (transformer). The extremely variable accuracy (between 5% and 95%) corresponds to classifying the validation set as all positive or all negative (orange validation, blue training).

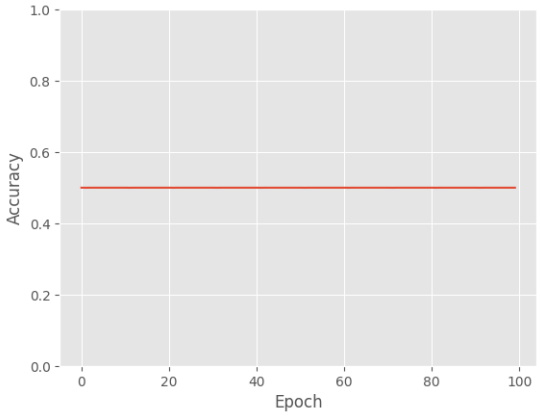

Figure 6.2: Training accuracy on downsampled graphs (GIN).

predict readmission. On individual downsampled graphs, the architecture was completely unable learn the training labels, or informative embeddings. When training the transformer, iterating over the training data simply switched the network from predicting all positive labels to all negative labels (6.1). On the GIN, the model was just randomly guessing (6.2). Worried that the models were perplexed by the randomly sampled training data, I also tried training on a fixed 50/50 split of positive and negative graphs - this had similarly poor results.

Due to the failure of the subgraph embedding model, the effectiveness of the similarity graph construction was not able to be tested.

6.1.1 What went wrong

As evidenced by [19], (homogeneous) transformer architecture is sufficient to model heterogeneous medical data. The lack of success must therefore be attributable to the input data being too unstructured. [19] does positional encoding on the discharge notes, and these have already been processed by a doctor to contain relevant information, in a manner from which other doctors ought to easily infer patient information - it is unsurprising that this processing makes it easier for models to infer importance of, and relationships between, medical terms, but less obvious a priori that a GNN couldn't do the same.

Chapter 7

Conclusions and Further Study

The negative result here is that completely unstructured EHR data is not sufficient as training data for a model predicting readmission. To improve upon this result, we require further study of how structured the data needs to be before readmission is learnable - this boundary lies somewhere between the "raw" approach here, and the expert-processed discharge summaries used in [19, 15]. A good start would be encoding temporality in the graph, or using the "importance" edge labels present in some of the MIMIC events tables, which I didn't use due to the additional complexity. If using temporal information, careful attention must be paid to ensure admission nodes can't "look into the future" and see another admission (of the same patient), to trivially predict readmission that way - a directed derived graph would suffice (with one-way message passing).

Due to the negative result, I was also not able to test the effectiveness of meta-path-based edges in the derived homogeneous graph. This motivates further experimentation on equivariant heterogeneous edge embedding methods.

We also saw that heterogeneous architecture is strictly more powerful than homogeneous GNN, with minimal additional computational overhead (needing only to store type information in memory), while still being equivariant. The 1-HWL algorithm (and resultant family of k-HWL algorithms) also requires study, in order to better understand the representational capacity of HGNN. The complexity of implementing full heterogeneous architecture was out of scope for this work. It would be interesting to see the outcome of "full" HGNN on the MIMIC graph.

Bibliography

[1] MIMIC-III documentation. https://mimic.mit.edu/docs/iii/.

[2] Emily Alsentzer, John R. Murphy, Willie Boag, Wei-Hung Weng, Di Jin, Tristan Naumann, and Matthew B. A. McDermott. Publicly available clinical BERT embeddings, 2019.

[3] Sebastiano Barbieri, James Kemp, Oscar Perez-Concha, Sradha Kotwal, Martin Gallagher, Angus Ritchie, and Louisa Jorm. Benchmarking deep learning architectures for predicting readmission to the icu and describing patients-at-risk. *Scientific reports*, 10(1):1111, 2020.

[4] Anas Belahcen, Monica Bianchini, and Franco Scarselli. Web spam detection using transductive (inductive graph neural networks. *Advances in Neural Networks: Computational and Theoretical Issues*, pages 83–91, 2015.

[5] Michael M. Bronstein, Joan Bruna, Taco Cohen, and Petar Veličković. Geometric deep learning: Grids, groups, graphs, geodesics, and gauges, 2021.

[6] Bruce G. Buchanan and Edward H. Shortliffe. *Rule Based Expert Systems: The Mycin Experiments of the Stanford Heuristic Programming Project (The Addison-Wesley Series in Artificial Intelligence)*. Addison-Wesley Longman Publishing Co., Inc., USA, 1984.

[7] Taco S. Cohen and Max Welling. Group equivariant convolutional networks, 2016.

[8] Tahani A Daghistani, Radwa Elshawi, Sherif Sakr, Amjad M Ahmed, Abdullah Al-Thwayee, and Mouaz H Al-Mallah. Predictors of in-hospital length of stay among cardiac patients: a machine learning approach. *International journal of cardiology*, 288:140–147, 2019.

[9] Zheng Dai, Siru Liu, Jinfa Wu, Mengdie Li, Jialin Liu, and Ke Li. Analysis of adult disease characteristics and mortality on MIMIC-III. *PloS one*, 15(4):e0232176, 2020.

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding, 2019.

[11] Yuxiao Dong, Nitesh V. Chawla, and Ananthram Swami. Metapath2vec: Scalable representation learning for heterogeneous networks. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, page 135–144, New York, NY, USA, 2017. Association for Computing Machinery.

[12] Elena Facco, Maria d'Errico, Alex Rodriguez, and Alessandro Laio. Estimating the intrinsic dimension of datasets by a minimal neighborhood information. *Scientific Reports*, 7(1), sep 2017.

[13] Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric, 2019.

[14] Thanos Gentimis, Alnaser Ala'J, Alex Durante, Kyle Cook, and Robert Steele. Predicting hospital length of stay using neural networks on mimic iii data. In *2017 IEEE 15th intl conf on dependable, autonomic and secure computing, 15th intl conf on pervasive intelligence and computing, 3rd intl conf on big data intelligence and computing and cyber science and technology congress (DASC/PiCom/DataCom/CyberSciTech)*, pages 1194–1201. IEEE, 2017.

[15] Sara Nouri Golmaei and Xiao Luo. DeepNote-GNN: predicting hospital readmission using clinical notes and patient network. In *Proceedings of the 12th ACM Conference on Bioinformatics, Computational Biology, and Health Informatics*, pages 1–9, 2021.

[16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[17] Priyanka Gupta, Pankaj Malhotra, Lovekesh Vig, and Gautam Shroff. Transfer learning for clinical time series analysis using recurrent neural networks. *arXiv preprint arXiv:1807.01705*, 2018.

[18] Silu He, Qinyao Luo, Ronghua Du, Ling Zhao, and Haifeng Li. STGC-GNNs: A GNN-based traffic prediction framework with a spatial-temporal granger causality graph, 2022.

[19] Kexin Huang, Jaan Altosaar, and Rajesh Ranganath. ClinicalBERT: Modeling clinical notes and predicting hospital readmission, 2020.

[20] Ningyuan Teresa Huang and Soledad Villar. A short tutorial on the weisfeiler-lehman test and its variants. In *ICASSP 2021 - 2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, jun 2021.

[21] A. Johnson, T. Pollard, and R Mark. Mimic-III clinical database (version 1.4). https://doi.org/10.13026/C2XW26, 2016.

[22] Alistair E.W. Johnson, Tom J. Pollard, Lu Shen, Li-wei H. Lehman, Mengling Feng, Mohammad Ghassemi, Benjamin Moody, Peter Szolovits, Leo Anthony Celi, and Roger G. Mark. MIMIC-III, a freely accessible critical care database. *Scientific Data*, 3(160035), 2016.

[23] Y. LECUN. The mnist database of handwritten digits. *http://yann.lecun.com/exdb/mnist/*, 1998.

[24] Yann LeCun, Bernhard Boser, John Denker, Donnie Henderson, R. Howard, Wayne Hubbard, and Lawrence Jackel. Handwritten digit recognition with a back-propagation network. In D. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 2. Morgan-Kaufmann, 1989.

[25] June-Goo Lee, Sanghoon Jun, Young-Won Cho, Hyunna Lee, Guk Bae Kim, Joon Beom Seo, and Namkug Kim. Deep learning in medical imaging: general overview. *Korean journal of radiology*, 18(4):570–584, 2017.

[26] Wenshuo Liu, Cooper Stansbury, Karandeep Singh, Andrew M Ryan, Devraj Sukul, Elham Mahmoudi, Akbar Waljee, Ji Zhu, and Brahmajee K Nallamothu. Predicting 30-day hospital readmissions using artificial neural networks with medical code embedding. *PloS one*, 15(4):e0221606, 2020.

[27] Zheng Liu, Xiaohan Li, Hao Peng, Lifang He, and Philip S. Yu. Heterogeneous similarity graph neural network on electronic health records, 2021.

[28] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013.

[29] Manish Motwani, Damini Dey, Daniel S Berman, Guido Germano, Stephan Achenbach, Mouaz H Al-Mallah, Daniele Andreini, Matthew J Budoff, Filippo Cademartiri, Tracy Q Callister, et al. Machine learning for prediction of all-cause mortality in patients with suspected coronary artery disease: a 5-year multicentre prospective registry analysis. *European heart journal*, 38(7):500–507, 2017.

[30] Kevin P. Murphy. *Probabilistic Machine Learning: An introduction*. MIT Press, 2022.

[31] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458*, 2015.

[32] Gihan Panapitiya, Michael Girard, Aaron Hollas, Jonathan Sepulveda, Vijayakumar Murugesan, Wei Wang, and Emily Saldanha. Evaluation of deep learning architectures for aqueous solubility prediction. *ACS omega*, 7(18):15695–15710, 2022.

[33] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.

[34] Alvin Rajkomar, Eyal Oren, Kai Chen, Andrew M Dai, Nissan Hajaj, Michaela Hardt, Peter J Liu, Xiaobing Liu, Jake Marcus, Mimi Sun, et al. Scalable and accurate deep learning with electronic health records. *NPJ digital medicine*, 1(1):18, 2018.

[35] Nils Reimers and Iryna Gurevych. Sentence-BERT: Sentence embeddings using siamese BERT-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 11 2019.

[36] Jonathan M Stokes, Kevin Yang, Kyle Swanson, Wengong Jin, Andres Cubillos-Ruiz, Nina M Donghia, Craig R MacNair, Shawn French, Lindsey A Carfrae, Zohar Bloom-Ackermann, et al. A deep learning approach to antibiotic discovery. *Cell*, 180(4):688–702, 2020.

[37] Siyi Tang, Amara Tariq, Jared Dunnmon, Umesh Sharma, Praneetha Elugunti, Daniel Rubin, Bhavik N Patel, and Imon Banerjee. Multimodal spatiotemporal graph neural networks for improved prediction of 30-day all-cause hospital readmission. *arXiv preprint arXiv:2204.06766*, 2022.

[38] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[39] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks?, 2019.

[40] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, jul 2018.

Appendix A

```
import pandas as pd
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
from torch.utils.data import Dataset, DataLoader, random_split, Subset
from torch_geometric.data import HeteroData
from sentence_transformers import SentenceTransformer
from torch_geometric.transforms import RandomLinkSplit, ToUndirected
from torch_geometric.nn import MetaPath2Vec
import torch_sparse
from torch_geometric.nn import GIN

path = "" #don't want to accidentally reveal my identity
path_to_mimic = "/mnt/sdd/MIMIC_data/mimic_3"

#generating readmission labels from MIMIC admissions table

df_adm = pd.read_csv(path_to_mimic + "/" + "ADMISSIONS.csv")
df_adm.ADMITTIME = pd.to_datetime(df_adm.ADMITTIME,
    format = '%Y-%m-%d %H:%M:%S', errors = 'coerce')
df_adm.DISCHTIME = pd.to_datetime(df_adm.DISCHTIME,
    format = '%Y-%m-%d %H:%M:%S', errors = 'coerce')
df_adm.DEATHTIME = pd.to_datetime(df_adm.DEATHTIME,
    format = '%Y-%m-%d %H:%M:%S', errors = 'coerce')
df_adm = df_adm.sort_values(['SUBJECT_ID','ADMITTIME'])
df_adm = df_adm.reset_index(drop = True)
df_adm['NEXT_ADMITTIME'] =
```

```
    df_adm.groupby('SUBJECT_ID').ADMITTIME.shift(-1)
df_adm['NEXT_ADMISSION_TYPE'] =
    df_adm.groupby('SUBJECT_ID').ADMISSION_TYPE.shift(-1)
rows = df_adm.NEXT_ADMISSION_TYPE == 'ELECTIVE'
df_adm.loc[rows,'NEXT_ADMITTIME'] = pd.NaT
df_adm.loc[rows,'NEXT_ADMISSION_TYPE'] = np.NaN
df_adm = df_adm.sort_values(['SUBJECT_ID','ADMITTIME'])
df_adm{[['NEXT_ADMITTIME','NEXT_ADMISSION_TYPE']]} =
    df_adm.groupby(['SUBJECT_ID'])
{[['NEXT_ADMITTIME','NEXT_ADMISSION_TYPE']]}.fillna('bfill')
df_adm['DAYS_NEXT_ADMIT']=
    (df_adm.NEXT_ADMITTIME -
    df_adm.DISCHTIME).dt.total_seconds()/(24*60*60)
df_adm1 = df_adm{[['SUBJECT_ID','HADM_ID','ADMITTIME',
    'DISCHTIME','DAYS_NEXT_ADMIT','NEXT_ADMITTIME',
    'ADMISSION_TYPE','DEATHTIME']]}
df_adm1['OUTPUT_LABEL'] =
    (df_adm1.DAYS_NEXT_ADMIT < 30).astype('int')
df_adm1=df_adm1{[["HADM_ID","SUBJECT_ID",
    "ADMITTIME","OUTPUT_LABEL"]]}
df_adm1 = df_adm1.sort_values(['HADM_ID'])
df_adm1 = df_adm1.reset_index(drop = True)
df_adm1.to_csv(path+"/CSVs/readmission_labels.csv")

#format events + codes tables

df_diag = pd.read_csv(
    "/mnt/sdd/MIMIC_data/mimic_3/DIAGNOSES_ICD.csv")
df_diag_icd = pd.read_csv(
    "/mnt/sdd/MIMIC_data/mimic_3/D_ICD_DIAGNOSES.csv")
df_diag_icd=
    df_diag_icd{[["ICD9_CODE","LONG_TITLE"]]}.astype(str)

# mapping diagnoses to index starting at 0

diag_map = { }
```

```
for i, name in enumerate(df_diag_icd.ICD9_CODE.unique()):
diag_map[name] = i

df_diag_icd["DIAG_ID"] =
    df_diag_icd["ICD9_CODE"].apply(lambda x: diag_map[x])
df_diag_icd =
    df_diag_icd.set_index("DIAG_ID",drop=True)
df_diag_icd.to_csv(path+"/CSVs/diag_codes.csv")

# apply same indexing to events table

df_diag["DIAG_ID"] =
    df_diag["ICD9_CODE"].apply(lambda x: ICD_dict(diag_map,x))
df_diag = df_diag.dropna()
df_diag.to_csv(path+"/CSVs/diagnoses.csv")

# unifying procedures tables (as there are two labelling systems annoyingly)

df_proc_codes_mv =
    pd.read_csv("/mnt/sdd/MIMIC_data/mimic_3/D_ITEMS.csv")
df_procs_mv = pd.read_csv(
    "/mnt/sdd/MIMIC_data/mimic_3/PROCEDUREEVENTS_MV.csv")
df_proc_codes_mv["combined"]=
    df_proc_codes_mv{['LABEL',"CATEGORY"]]}
    .astype(str).agg(' '.join, axis=1)
df_proc_codes_mv =
    df_proc_codes_mv{["ITEMID","combined"]]}
df_proc_codes_mv.to_csv(
    path+"/CSVs/proc_codes_mv.csv")
df_procs_mv =
    df_procs_mv{['SUBJECT_ID', 'HADM_ID','ITEMID']]}
df_procs_mv.to_csv(
    path+"/CSVs/procs_mv.csv")
df_procs_icd = pd.read_csv(
```

```
    "/mnt/sdd/MIMIC_data/mimic_3/PROCEDURES_ICD.csv")
df_procs_icd.to_csv(path+"/CSVs/procs_icd.csv")
df_proc = pd.read_csv(
    "/mnt/sdd/MIMIC_data/mimic_3/D_ICD_PROCEDURES.csv")
df_proc = df_proc{[["ICD9_CODE","LONG_TITLE"]]}
df_proc.to_csv(path+"/CSVs/proc_codes_icd.csv")

#patient table - used connectivity, but sadly never used demographics

df_pat = pd.read_csv("/mnt/sdd/MIMIC_data/mimic_3/PATIENTS.csv")
df_pat= df_pat{[['SUBJECT_ID', 'GENDER', 'DOB']]}
df_pat.to_csv(path+"/CSVs/demographics.csv")

#combining MV and CV procedures

df_procs_icd = pd.read_csv(path+"/CSVs/procs_icd.csv")
df_procs_mv = pd.read_csv(path+"/CSVs/procs_mv.csv")
df_procs_icd = df_procs_icd{[['SUBJECT_ID', 'HADM_ID','ICD9_CODE']]}
df_procs_mv = df_procs_mv{[['SUBJECT_ID', 'HADM_ID', 'ITEMID']]}
df_comb =
    pd.concat((df_procs_icd.rename(
    columns={"ICD9_CODE":"ITEMID"}),df_procs_mv))
df_proc_codes_icd =
    pd.read_csv(path+"/CSVs/proc_codes_icd.csv")
df_proc_codes_mv =
    pd.read_csv(path+"/CSVs/proc_codes_mv.csv")
df_codes_comb =
    pd.concat((df_proc_codes_icd.rename(columns=
    {"ICD9_CODE":"ITEMID","LONG_TITLE":"combined"}),
    df_proc_codes_mv)).rename(columns={"combined":"LONG_TITLE"})
    {[["ITEMID","LONG_TITLE"]]}
df_comb.to_csv(path+"/CSVs/procs_combined.csv")
df_codes_comb.to_csv(path+"/CSVs/proc_codes_combined.csv")

#convert pd dfs to tensor dataset (using clinicalBERT+sBert)
```

```
def load_node_csv(path, index_col, encoders=None, **kwargs):
df = pd.read_csv(path, index_col=index_col, **kwargs)
mapping = {index: i for i, index in enumerate(df.index.unique())}

x = None
if encoders is not None:
xs = [encoder(df[col]) for col, encoder in encoders.items()]
x = torch.cat(xs, dim=-1)

return x, mapping

class SequenceEncoder(object):
def __init__(self, model_name="emilyalsentzer/Bio_ClinicalBERT", device=None):
self.device = device
self.model = SentenceTransformer(model_name, device=device)

@torch.no_grad()
def __call__(self, df):
x = self.model.encode(df.values, show_progress_bar=True,
convert_to_tensor=True, device=self.device)
return x.cpu()

SE = SequenceEncoder(device = device)

def save_dict(dic, address):
try:
geeky_file = open(address, 'wb')
pickle.dump(dic, geeky_file)
geeky_file.close()

except:
print("Something went wrong")

# making sure that indexing is all proper (across multiple files)
# and generating sBERT encoded tensors
```

```
diag_codes, diag_mapping =
    load_node_csv(path+"/CSVs/diag_codes.csv","DIAG_ID",{"LONG_TITLE":SE})

save_dict(diag_mapping,path+"/Features/diag_mapping.pkl")

torch.save(diag_codes,path+"/Features/diag_codes.pt")

proc_codes, proc_mapping =
    load_node_csv(
    path+"/CSVs/proc_codes_combined.csv","ITEMID",{"LONG_TITLE":SE})

save_dict(proc_mapping,path+"/Features/proc_mapping.pkl")

torch.save(proc_codes,path+"/Features/proc_codes.pt")

pres_codes, pres_mapping =
    load_node_csv(
    path+"/CSVs/pres_codes.csv","DRUG_ID",{"DRUG":SE})

save_dict(pres_mapping,path+"/Features/pres_mapping.pkl")

torch.save(pres_codes,path+"/Features/pres_codes.pt")

_, visit_mapping = load_node_csv(
    path+"/CSVs/readmission_labels.csv","HADM_ID")

_, patient_mapping = load_node_csv(
    path+"/CSVs/readmission_labels.csv","SUBJECT_ID")

save_dict(visit_mapping,path+"/Features/visit_mapping.pkl")

save_dict(patient_mapping,path+"/Features/patient_mapping.pkl")

# creating heterodata object
#both to get a handle on the data, and to do MP2Vec
```

```
data= HeteroData()
# loading in nodes
data[“procs”].x = proc_codes
data[“diags”].x = diag_codes
data[“pres”].x = pres_codes

def load_edge_csv(path, src_index_col, src_mapping,
    dst_index_col, dst_mapping, encoders=None, **kwargs):
df = pd.read_csv(path, **kwargs)

src = [src_mapping[index] for index in df[src_index_col]]
dst = [dst_mapping[index] for index in df[dst_index_col]]
edge_index = torch.tensor([src, dst])

edge_attr = None
if encoders is not None:
edge_attrs = [encoder(df[col]) for col, encoder in encoders.items()]
edge_attr = torch.cat(edge_attrs, dim=-1)

return edge_index, edge_attr

# and loading in edges (no edge features)

pres_edge_index, _ = load_edge_csv(path+“/CSVs/pres_edges.csv”,
    “HADM_ID”,visit_mapping,“DRUG_ID”,pres_mapping)

diag_edge_index, _ = load_edge_csv(
    path+“/CSVs/diagnoses.csv”,
    “HADM_ID”,visit_mapping,“DIAG_ID”, diag_mapping)

proc_edge_index, _ = load_edge_csv(
    path+“/CSVs/procs_combined.csv”,
    “HADM_ID”, visit_mapping, “ITEMID”, proc_mapping)

admissions_edges_index , _ = load_edge_csv(
```

```
    path+"/CSVs/readmission_labels.csv",
    "HADM_ID", visit_mapping, "SUBJECT_ID", patient_mapping)

torch.save(pres_edge_index,path+"/Features/pres_edge_index.pt")

torch.save(diag_edge_index,path+"/Features/diag_edge_index.pt")

torch.save(proc_edge_index,path+"/Features/proc_edge_index.pt")

data["admission", "prescribed", "pres"].edge_index = pres_edge_index

data["admission", "diagnosed", "diags"].edge_index = diag_edge_index

data["admission", "had_procedure", "procs"].edge_index = proc_edge_index

data["admission", "is_patient", "patients"].edge_index = admissions_edges_index

# load in readm labels

readm = pd.read_csv(path+"/CSVs/readmission_labels.csv")
    {[["HADM_ID", "OUTPUT_LABEL"]]}

readm["IDX"] = readm["HADM_ID"].apply(lambda x: visit_mapping[x])

values = np.array(readm["OUTPUT_LABEL"].values)
n_values = np.max(values) + 1
data["admission"].y = torch.Tensor(np.eye(n_values)[values])

data = ToUndirected()(data)


#do MP2Vec admission node embedding
# MetaPath2Vec() from PyG - tell model which metapaths to use

MP_adm_diag = [("admission","diagnosed","diags"),
```

```
    ('diags', 'rev_diagnosed', 'admission')]

MP_adm_pres = [('admission', 'prescribed', 'pres'),
    ('pres', 'rev_prescribed', 'admission')]

model = MetaPath2Vec(data.edge_index_dict,
    embedding_dim=64, metapath=MP_adm_diag).to(device)

loader = model.loader(batch_size=128, shuffle=True, num_workers=6)
optimizer = torch.optim.SparseAdam(list(model.parameters()), lr=0.01)

# need to train model to do skip-gram embedding

def train(epoch, log_steps=100, eval_steps=2000):
model.train()

total_loss = 0
for i, (pos_rw, neg_rw) in enumerate(loader):
optimizer.zero_grad()
loss = model.loss(pos_rw.to(device), neg_rw.to(device))
loss.backward()
optimizer.step()

total_loss += loss.item()
if (i + 1) % log_steps == 0:
print((f'Epoch: {epoch}, Step: {i + 1:05d}/{len(loader)}, '
f'Loss: {total_loss / log_steps:.4f}'))
total_loss = 0

for epoch in range(1, 200):
train(epoch,log_steps = 100)

model.eval()
#get embeddings (patient metapath)

adm_diags = model("admission")
```

```
torch.save(adm_diags,path+“/Tensors/” + “adm_diags_64.pt”)

model = MetaPath2Vec(data.edge_index_dict, embedding_dim=64,
metapath=MP_adm_pres).to(device)

for epoch in range(1, 100):
train(epoch,log_steps = 100)

model.eval()
#get embeddings (diagnoses metapath)

adm_pres = model(“admission”)
torch.save(adm_pres,path+“/Tensors/” + “adm_pres_64.pt”)

MP_adm = torch.cat((adm_diags,adm_pres),1)
torch.save(MP_adm,path+“/Tensors/” + “adm_MP_128.pt”)

#get adj from MP embeddings

adm_feats = torch.load(path+“/Tensors/adm_MP_64.pt”)

norm_adm = F.normalize(adm_feats).to(“cpu”)
sim_mat =(norm_adm @ norm_adm.T)
sim_mat -= torch.diag(torch.diag(sim_mat))
torch.save(U.dense_to_sparse(sim_mat>0.9)[0],
    path+“/Tensors/sparse_adj_90.pt”)
torch.save(U.dense_to_sparse(sim_mat>0.8)[0],
    path+“/Tensors/sparse_adj_80.pt”)

#set up functions to get graph (tensor) for given admission

diag_codes = torch.load(path+“/”+“diag_codes”+“.pt”)
pres_codes = torch.load(path+“/”+“pres_codes”+“.pt”)
proc_codes = torch.load(path+“/”+“proc_codes”+“.pt”)
diag_mapping = pickle.load(
```

```
    open(path + “/” + “diag” + “_mapping.pkl”,“rb”))
pres_mapping = pickle.load(
    open(path + “/” + “pres” + “_mapping.pkl”,“rb”))
proc_mapping = pickle.load(
    open(path + “/” + “proc” + “_mapping.pkl”,“rb”))
patient_mapping = pickle.load(
    open(path + “/” + “patient” + “_mapping.pkl”,“rb”))
visit_mapping = pickle.load(
    open(path + “/” + “visit” + “_mapping.pkl”,“rb”))

proc_edge_index = torch.load(path+“/”+“proc_edge_index”+“.pt”)
diag_edge_index = torch.load(path+“/”+“diag_edge_index”+“.pt”)
pres_edge_index = torch.load(path+“/”+“pres_edge_index”+“.pt”)
readm_labels = pd.read_csv(path+“/CSVs/readmission_labels.csv”)
readm_labels = readm_labels{[[“HADM_ID”, “OUTPUT_LABEL”]]}

class TensorLoader(object):# class to load subgraph tensors
def __init__(self):
path = path+“/Features”
self.diag_codes = torch.load(path+“/”+“diag_codes”+“.pt”)
self.pres_codes = torch.load(path+“/”+“pres_codes”+“.pt”)
self.proc_codes = torch.load(path+“/”+“proc_codes”+“.pt”)
self.diag_mapping = pickle.load(
    open(path + “/” + “diag” + “_mapping.pkl”,“rb”))
self.pres_mapping = pickle.load(
    open(path + “/” + “pres” + “_mapping.pkl”,“rb”))
self.proc_mapping = pickle.load(
    open(path + “/” + “proc” + “_mapping.pkl”,“rb”))
#self.patient_mapping = pickle.load(
    open(path + “/” + “patient” + “_mapping.pkl”,“rb”))
self.visit_mapping = pickle.load(
    open(path + “/” + “visit” + “_mapping.pkl”,“rb”))
self.proc_edge_index = torch.load(path+“/”+“proc_edge_index”+“.pt”)
self.diag_edge_index = torch.load(path+“/”+“diag_edge_index”+“.pt”)
self.pres_edge_index = torch.load(path+“/”+“pres_edge_index”+“.pt”)
self.readm_labels = pd.read_csv(path+“/CSVs/readmission_labels.csv”)
```

{[[“HADM_ID”, “OUTPUT_LABEL”]]}

```
def get_tensor(self,hadm_id):
visit_id = self.visit_mapping[hadm_id]

proc_indices = self.proc_edge_index[0]==visit_id
proc_items = self.proc_edge_index[1][proc_indices].unique()
proc_tensor = self.proc_codes[proc_items]

diag_indices = self.diag_edge_index[0]==visit_id
diag_items = self.diag_edge_index[1][diag_indices].unique()
diag_tensor = self.diag_codes[diag_items]

pres_indices = self.pres_edge_index[0]==visit_id
pres_items = self.pres_edge_index[1][pres_indices].unique()
pres_tensor = self.pres_codes[pres_items]

items_tensor = torch.cat((proc_tensor,diag_tensor,pres_tensor),0)

# print(f“{proc_tensor.shape=}”)
# print(f“{diag_tensor.shape=}”)
# print(f“{pres_tensor.shape=}”)
# print(f“{items_tensor.shape=}”)

return items_tensor

TL = TensorLoader()

#get training labels (for given index)
pos_indices = list(TL.readm_labels[TL.readm_labels.OUTPUT_LABEL==1].index)
neg_indices = [x for x in TL.readm_labels.index if x not in pos_indices]

class TensorDataset(Dataset): # dataset where each datum is subgraph
def __init__(self):
self.TL = TensorLoader()
```

```
self.labels = self.TL.readm_labels

def __len__(self):
return len(self.labels)

def __getitem__(self,idx):
row = self.labels.iloc[idx]
tensor = self.TL.get_tensor(row["HADM_ID"])
return tensor, torch.Tensor([1 if i == row["OUTPUT_LABEL"] else 0 for i in [0,1]])

#to train transformer on more than one datum at a time
def collate_fn_pad(batch):
lengths = torch.Tensor([t[0].shape[0] for t in batch])
sequence = [t[0] for t in batch]
ys = [torch.unsqueeze(t[1],0) for t in batch]
padded = torch.nn.utils.rnn.pad_sequence(sequence, True)
mask = padded != 0
return (padded,torch.cat(ys)), lengths, mask[:,:,0]

train_dl = DataLoader(td, batch_size = 16,collate_fn = collate_fn_pad)

class single_readm_transformer(nn.Module):

def __init__(self):
super().__init__()
encoder_layer = torch.nn.TransformerEncoderLayer(768, n_heads=8,
    dim_feedforward=768,batch_first=True)
self.layer = torch.nn.TransformerEncoder(
    encoder_layer, num_layers=6, mask_check=True)

self.token = nn.Parameter(torch.rand((1,1,768)))
self.MLP = nn.Sequential(nn.Linear(768,2)) #no softmax for logits loss

def forward(self, x, mask):
#mask shape is x.shape[0], x.shape[1].
# records whether that row is padding or not
```

```
#could use lengths instead here as well
out = self.get_class_token(x, mask)
#i.e. get embedding for subgraph, then apply linear layer
return self.MLP(out)


def get_class_token(self, x, mask): #BUG FOUND! need to take NOT of the mask?
token = self.token.expand((x.shape[0],1,768))
x_data = torch.cat((token.to(device), x),1)

token_mask = torch.ones((x.shape[0],1)).bool()
mask_data = torch.cat((token_mask.to(device), mask),1)
out = self.layer(x_data, src_key_padding_mask = torch.logical_not(mask_data))
return out[:,0]

#training function
from sklearn.metrics import accuracy_score

def train(net, dataloader, optim, loss_func, epoch):
net.train() # put network in train mode
total_loss = 0
preds = []
trues = []
batches = 0

for idx, ((data, target), lengths, mask) in enumerate(dataloader):
if batches ==0:
start = time.time()

#load data for this epoch
#data,target = Variable(data), Variable(target) # I think this is deprecated
batches += 1

#do training
net.zero_grad()
optim.zero_grad()
```

```
#forward pass
pred = net(data.to(device), mask.to(device))
loss = loss_func(pred,target.to(device))

#backward pass
loss.backward()
optim.step()
total_loss+=loss

#bookkeeping - everything is one-hot so this makes it easier to calc accuracy
preds.append(np.argmax(pred.detach().cpu().numpy(), axis=1).reshape(-1))
trues.append(np.argmax(target.detach().cpu().numpy(), axis=1).reshape(-1))

if idx % 100 == 0: #Report stats every x batches
print('Train Epoch: { } [{ }/{ } ({:.0f}%)]
    \tLoss: {:.6f}'.format(epoch, (idx+1) * len(data), len(dataloader.dataset),
    100. * (idx+1) / len(dataloader), loss.item()), flush=True)
if batches==1:
end_time = time.time()
time_lapsed = end_time - start
print(f"{time_lapsed=}")

#calculate summary stats
av_loss = total_loss/batches
av_loss = av_loss.detach().cpu().numpy()

preds = np.concatenate(preds).reshape(-1)
trues = np.concatenate(trues).reshape(-1)
acc = accuracy_score(preds, trues)

return av_loss, acc


def val(net, val_dataloader, optim, loss_func, epoch):
net.eval() #Put the model in eval mode
```

```
total_loss = 0
preds = []
trues = []
batches = 0

with torch.no_grad(): # So no gradients accumulate
for idx, ((data, target), lengths, mask) in enumerate(val_dataloader):
batches+=1
#data, target = Variable(data), Variable(target)

#forward pass
pred = net(data.to(device), mask.to(device))
loss = loss_func(pred,target.to(device))

total_loss += loss
preds.append(np.argmax(pred.detach().cpu().numpy(), axis=1).reshape(-1))
trues.append(np.argmax(target.detach().cpu().numpy(), axis=1).reshape(-1))

#compute summary stats
av_loss = total_loss/batches
av_loss = av_loss.detach().cpu().numpy()

preds = np.concatenate(preds).reshape(-1)
trues = np.concatenate(trues).reshape(-1)
acc = accuracy_score(preds, trues)

print('Validation set: Average loss: {:.4f}'.format(av_loss, flush=True))
print('Validation set: Average Acc: {:.4f}'.format(acc, flush=True))
print('\n')

return av_loss, acc



#getting ready to train!
net = single_readm_transformer().to(device)
class_loss = nn.BCEWithLogitsLoss().to(device)
```

```
optim = torch.optim.Adam(net.parameters())
losses = []

#training loop for transformer
max_epochs = 30
for epoch in range(1, max_epochs+1):
train_dataloader = DataLoader(
    Subset(td,pos_indices+random.sample(neg_indices,3227)),
    batch_size = 16,collate_fn = collate_fn_pad, shuffle=True)
train_loss, train_acc = train(net, train_dataloader, optim, class_loss, epoch)
val_loss, val_acc = val(net, val_dataloader, optim, class_loss, epoch)
losses.append([train_loss, train_acc, val_loss, val_acc])



#training GIN
X_train = Subset(td,pos_indices+random.sample(neg_indices,3227))

#preprocessing + batching
batch_size = 4
A = []
X = []
Y = []
for x,y in X_train:
X_shape = x.shape[0]
adj_matrix = torch.ones((X_shape,X_shape))
A.append(adj_matrix)
X.append(x)
Y.append(y[0:1])

from scipy.linalg import lapack
import math
def graph_mini_batch(adj_matrix_list, x_list, y_list, batch_size=64):
assert(len(adj_matrix_list)==len(x_list) and len(x_list)==len(y_list))
length_left=len(x_list)
iters_needed = math.ceil(length_left/batch_size)
```

```
for i in range(iters_needed):
if length_left>=batch_size:
this_batch=batch_size
else:
this_batch=length_left
length_left -= this_batch
#this_batch tells us how many things we're going to stick together now
start_index = i*batch_size
adj_round = adj_matrix_list[start_index:start_index+this_batch]
x_round = x_list[start_index:start_index+this_batch]
y_round = y_list[start_index:start_index+this_batch]

#var_round vars hold list of relevant tensors for this round of iter
sizes = [x.shape[0] for x in x_round]
batch_bits = []
for j in range(this_batch):
left_sum = sum(sizes[0:j])
right_sum= sum(sizes[j+1:this_batch])
tens = adj_round[j] # tens is the tensor at position j to stick bits onto the ends of
tens_size = tens.shape[0]
#MATHS: need dim0 to refer to size of tens, dim1 to refer to left and right sum
left_block = torch.zeros(tens_size,left_sum)
right_block = torch.zeros(tens_size,right_sum)
# print(f"left dim: {(tens_size,left_sum)},
    right dim: {(tens_size,right_sum)},
    middle dim: {tens_size},
    block dim: {tens_size+left_sum+right_sum}")
adj_round[j] = torch.cat((left_block,tens,right_block),dim=1)
batch_bit = torch.ones(tens_size)
batch_bit = torch.mul(batch_bit,j+1)
batch_bits.append(batch_bit)
#stuck stuff onto adj_round tensors
A_B = torch.cat(adj_round)
X_B = torch.cat(x_round)
Y_B = torch.cat(y_round)
Batch = torch.cat(batch_bits)
```

```
yield (A_B,X_B,Y_B,Batch)

#function to get graph embedding from node embeddings
def global_sum_pool(x, batch):
return scatter(x,batch.to(torch.int64).add(-1),0)

params = {
“input_features”: 768,
“hidden_features”: 50,
“num_layers”: 4,
“learning_rate”: 1e-4,
“weight_decay”: 0,
“num_epochs”: 100,
“num_classes”: 2,
“batch_size”: 4
}

gin = GIN(params[“input_features”],
    params[“hidden_features”], params[“num_layers”]).to(device)

losses = []
accuracies = []
begin_time = time.time()
pooling = global_sum_pool
graph_pred_linear = torch.nn.Linear(
    params[“hidden_features”], params[“num_classes”]).to(device)
loss_fn = nn.CrossEntropyLoss().to(device)
model_param_group =
    [{“params”: gin.parameters(), “lr”: params[“learning_rate”]}]
if graph_pred_linear is not None:
model_param_group.append(
    {“params”: graph_pred_linear.parameters(), “lr”: params[“learning_rate”]} )

optimizer = torch.optim.AdamW(model_param_group,
    lr=params[“learning_rate”],
```

```
    weight_decay=params["weight_decay"])
for epoch in range(params["num_epochs"]):
gin.train()
ep_loss = 0
for adj_matrix, x, y, batch in
    graph_mini_batch(A, X, Y, params["batch_size"]):
optimizer.zero_grad()
edge_index = U.dense_to_sparse(adj_matrix)[0]
nodes = gin(x.to(device), edge_index.to(device))
graph_reps = pooling(nodes, batch.to(device))
pred = graph_pred_linear(graph_reps)
loss = loss_fn(pred, y.to(device).long())
ep_loss+=loss
loss.backward()
optimizer.step()
if epoch % 10 == 0:
gin.eval()
correct = 0
total_num = 0
for adj_matrix, x, y, batch in
    graph_mini_batch(A, X, Y, params["batch_size"]):
optimizer.zero_grad()
edge_index = U.dense_to_sparse(adj_matrix)[0]
nodes = gin(x.to(device), edge_index.to(device))
graph_reps = pooling(nodes, batch.to(device))
pred = graph_pred_linear(graph_reps).detach().cpu()
correct += (pred.argmax(dim=-1) == y).sum()
total_num += len(y)
accuracies.append(correct/total_num)
print("epoch={}, loss={ }, accuracy={}"
    .format(epoch, loss.item(), correct/total_num))
losses.append(ep_loss.detach().cpu()/params["batch_size"]/params["num_epochs"])
print("time={}".format(time.time()-begin_time))
```

Thanks for reading!