UNIVERSITÀ DI PISA

DIPARTIMENTO DI MATEMATICA

Corso di Laurea Magistrale in Matematica

Predicting and Explaining on Electronic Health Records with Transformer-based Models

Relatori:

Roberto Pellungrini

Fosca Giannotti

Candidato:

Alessio Marchetti

ANNO ACCADEMICO 2022/2023

Contents

1 Introduction

Rapid growth of information systems in the healthcare sector has made large amounts of relevant data available in the last decades. One very important and rich form of this kind of data is the Electronic Health Records (EHR), among which we can find sequences of visit descriptions taken at irregular time distances. Visits can consist in informations on the medical conditions, symptoms, diagnoses, medications and actions taken during the admission.

With large amounts of data available, a significant opportunity for the advancement of the quality of the services offered is presented through its analysis. One important task is the prediction of future analysis given the patient history in the form of EHR. This is a nontrivial task given the high dimensionality of the problem and its temporal dependencies. However several approaches in the field of deep learning have been proposed to tackle this task, mainly based on recurrent architectures [8, 18, 17] while an exploration transformer-based architectures has just started [22].

Good predictive models are not enough though. Models as critical as those in the healthcare sphere have to work in tandem with professionals. To increase the trustworthiness of the models, predictions should not only accurate but also understandable by humans. However, inner logic of state-of the-art deep-learning models is still poorly understood. Work has been conducted in the last years to work around this limit, an important example being doctorXAI [20], an agnostic explainable model specifically built to leverage ontological information. Its core idea is to approximate the local behaviour of a opaque model with an intrinsically explainable predictor.

In this thesis we want to improve and fully understand the capabilities of doctorXAI. Our aim is to improve the predictive accuracy of the underlying model and to provide more insight into the explanations provided by doctorXAI, to improve its usability and effectiveness for physicians. Thus we work in several directions.

1. *On learning a black box model:* We substitute the underlying recurrent black-box model with a transformer-based one, trained on an updated version of the dataset. This allows for better performances in the predictions. Our new model will also allow us to extract some inner data (the attention values) and associate it to portions of the input to analyze the inner workings of the model.

2. *On generating explanations:* We propose a new method for generating a synthetic neighborhood that should be aware of the underlying data distribution. This would allow for a more thorough exploration of the local behavior of the predictive model. The process would be

done by adding noise to real data and then extracting samples over a distribution for the possible reconstruction given by a deep learning model.

3. *On using the explanation for model investigation:* We try to explore some parts of the transformer black-box model through the lens of the local explanation. In particular, we compute the correlation between the attention score given by the self-attention heads in the transformer and the feature importance given by the tree in the local explanation pipeline. We will find out that the correlation is very weak and attention is not a good proxy for the explanation.

2 Related

2.1 LLM and Transformers

The development of predictor models for categorical sequential data has historically been deeply intertwined with text generation and machine-translation tasks. In the most common approaches, the text is divided into smaller portions called tokens, which usually represent a word, a suffix, a punctuation symbol, or an indivisible part of the language. Typical values for the number of tokens in a single language are in the order of a few tens of thousands.

The text generation task is the problem of generating a text answer to a text prompt, which can be a question or some kind of instruction. The machine translation task is the problem of generating the translation to some given input text. These two tasks are very similar, and for clarity, we will focus on the translation one in the following discussion.

Handling sequence of data of arbitrary length has always been a challenge in the context of deep learning, due to the fixed size of the input and output layers in Neural Networks. The first effective solution has been the idea of recurrent architectures.

The task is split in two parts: in the first part the model gets in input the tokens of the prompt to translate. In the second part the answer is generated in an autoregressive manner: starting from a special `<start>` token, the model generates the next token. The new token is then concatenated to the previous ones, and is given to the model, which generates a next token. This process continues until a special `<end>` token is generated, or the generated sequence length exceeds a fixed limit.

2.1.1 Recurrent Models and Attention

A recurrent model is built in such a way that it can take in input a hidden memory vector h, and a token t, and then outputs h', an updated version of h. The new hidden vector is then fed to the model together with the next token [9].

The process is started with a fixed initial vector h_0, and let's assume the prompt is made by the token sequence $t_0, \dots, t_n$. Then given the recurrent model f, the first hidden vector generated is $h_1 = f(h_0, t_0)$, and in general we define $h_(i + i) = f(h_i, t_i)$ for $i = 1 \dots n$. This tackles the first part of the problem.

A very simple form for the model f is the following. Let $e_1, \dots, e_N$ be a sequence of fixed k-dimensional vectors (these are called embeddings), where N is the number of possible tokens and k is a fixed integer. Let σ be a

non linear activation function, typical choices are the Rectified Linear Unit (ReLU) or the sigmoid function. Let W_h and W_t be two $n \times n$ matrices a b a k-dimensional vector. Then if h is a k-dimensional hidden state, and t a token corresponding to the vector e_j, we set $f(h,t) = \sigma(W_h h + W_t e_j + b)$.

Other more complex architectures has been explored, but this simple presented one is the base for all of them.

For the second part we start with the `<start>` token which we will call g_0. From this we generate a new hidden state $\hat{h}_1 = f(h_{n+1}, g_0)$. With a classical NN model f_{gen} we can use $\hat{h}_1$ to generate a probability distribution $f_{\text{gen}}(\hat{h}_1)$ over all the possible tokens. We can then extract a new token from this distribution. Usual approaches consists in taking the maximum probability token, a plain sampling over the distribution, or some hybrid methods in which the sample is performed only on the most probable tokens. The result of this process is a new token g_1. From this point onward we repeat the process, generating $\hat{h}_2 = f(\hat{h}_1, g_1)$ and from the new hidden state a new token, and so on. An example of this architecture is shown in Figure 1.

The training of this kind of models is performed noticing that it defines a probability distribution over all the possible answer sentences. In fact each new token is generated from a probability distribution given the previous ones. This is effectively a decomposition of the probability of the whole sentence. Thus the training is performed maximizing the likelihood of a dataset of human generated pairs of input and outputs.

As in most of trainings of Neural Networks, the optimization is done through Stochastic Gradient Descent and backpropagation. This reveals the biggest weakness of recurrent architectures. Assuming the model has dealt with n tokens between inputs and outputs, which means that reasonable values for n could range from a few dozens to few hundreds or even more, the gradient for the first application of f has to go through all the n layers that follow to arrive to the loss. During all these steps the quality of the gradient is degraded, usually leading to the phenomenons of exploding gradients or vanishing gradients [21]. The idea is that the flow of the gradient is a dynamical system which can be mostly described by a multiplication by a matrix. If these matrices consistently increase the magnitude of the gradient, the gradient will become very large, and thus a step of Stochastic Gradient Descent (SGD) will move the model parameters in a region in which the function is not approximated by the gradient in the starting point. This means that the update is meaningless. If instead the matrices consistently decrease the magnitude of the gradient, the update of the SGD will be near zero, and the model will fail to converge.

Several attempts to solve this problem have been tried by modifying the inner architecture of the recurrent net, and the most notable examples of that

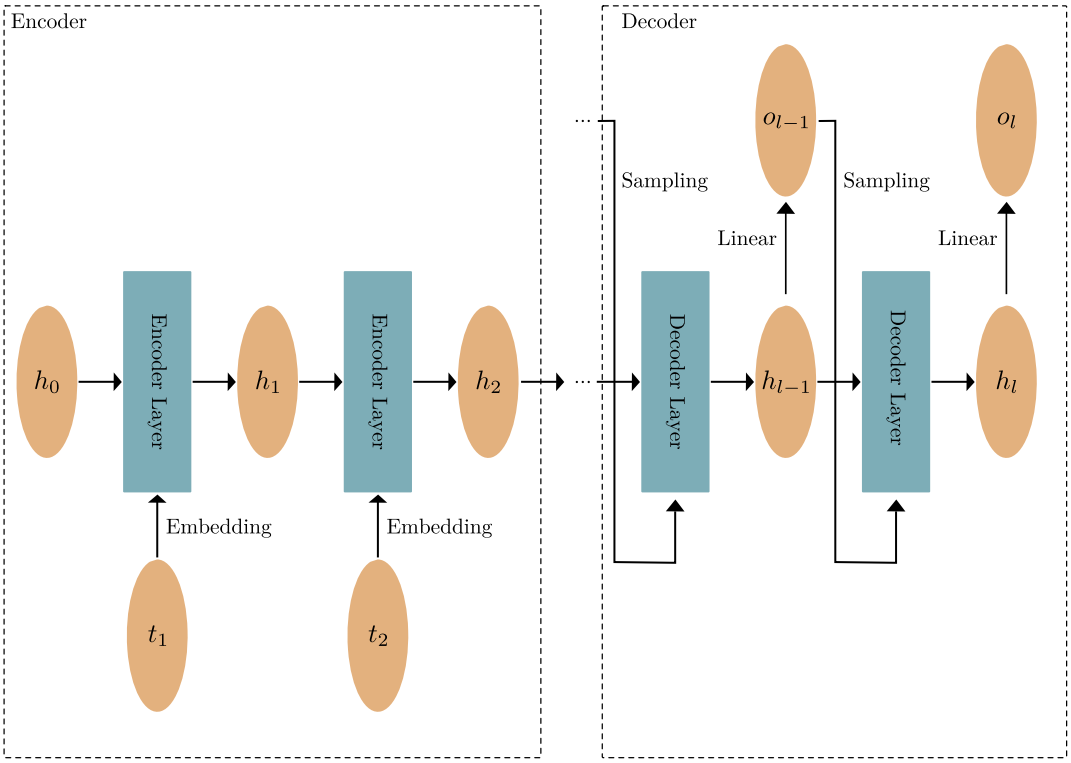

Figure 1: Example of a simple RNN architecture.

are the Gate Recurrent Units (GRUs) and the Long Short Term Memory models (LSTMs) [9]. These are however only a mitigation of the problem, and not a complete solution.

One first possible modification, that does not solve the gradient problem but still improves the performances of the model is using two different models for the two parts of the task. These two models have the same structure of the previous recurrent ones, but they are used for two different objectives. The first part of the problem is to encode the input tokens into a single hidden vector, and thus the model is called encoder. The second part of the problem is to take the hidden vector and generate autoregressively the output, in fact decoding the vector into tokens. This second model is thus called decoder.

A very important step for the evolution of these architectures has been in noticing that the hidden state during the decoder phase of the process has two distinct functions: remembering the input, and remembering at which point of the output generation the model is at. This suggest that the two functions could be carried out by two different vectors. This can be easily

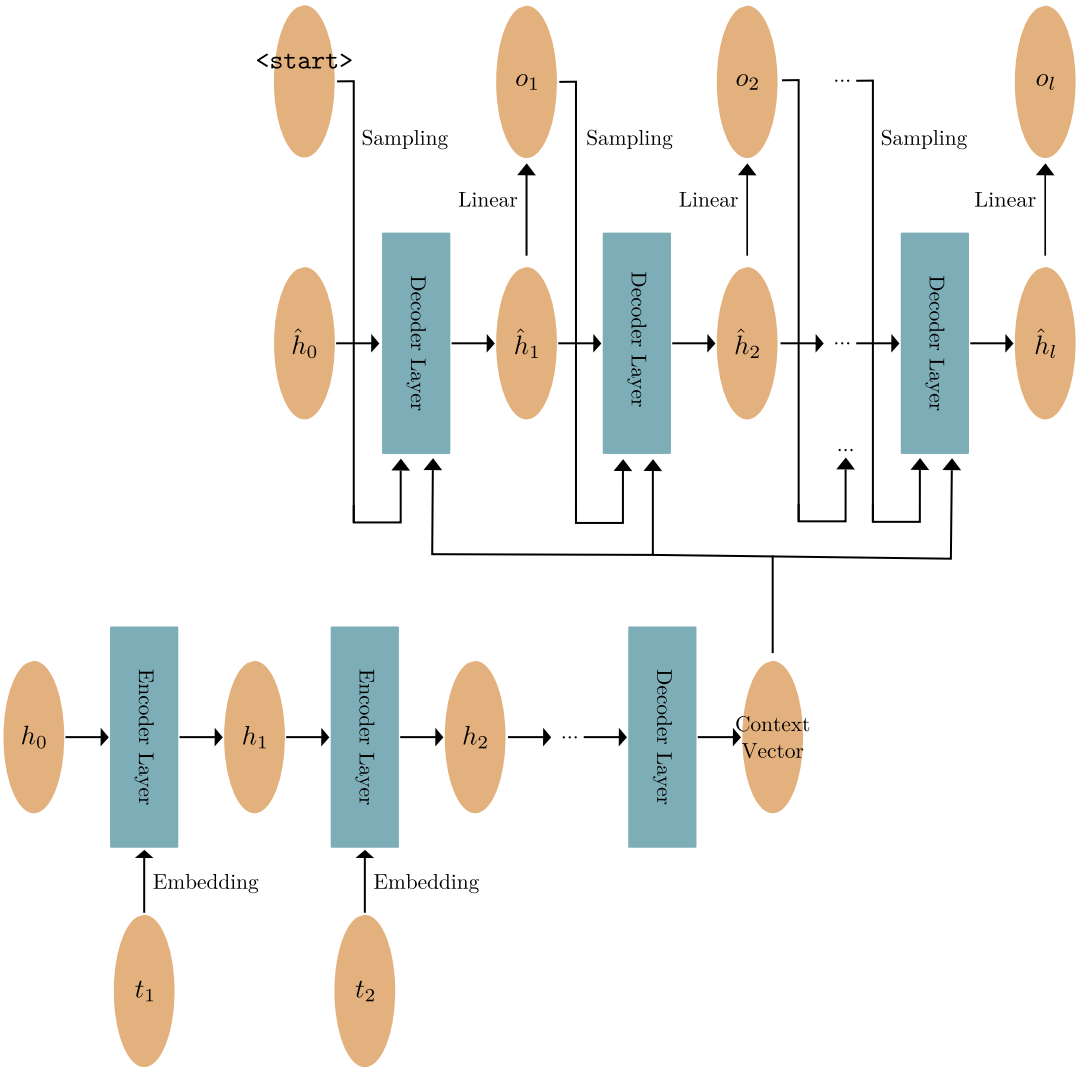

Figure 2: An example of a recurrent architecture with a context vector.

accomplished by having the decoder model be a function of a decoder hidden state, a context vector, which is the final output of the encoder, and the previous token. The result is that the decoder is unable to forget the encoder output, because it is not modified during the recurrence. An example of this kind of architecture is shown in figure 2.

The next step has been improving this mechanism. In the translation task, it's very common for a word in the output to map directly to a single word in the input, and more generally small groups of words in the outputs correspond to small groups of words in the input. On the other hand most

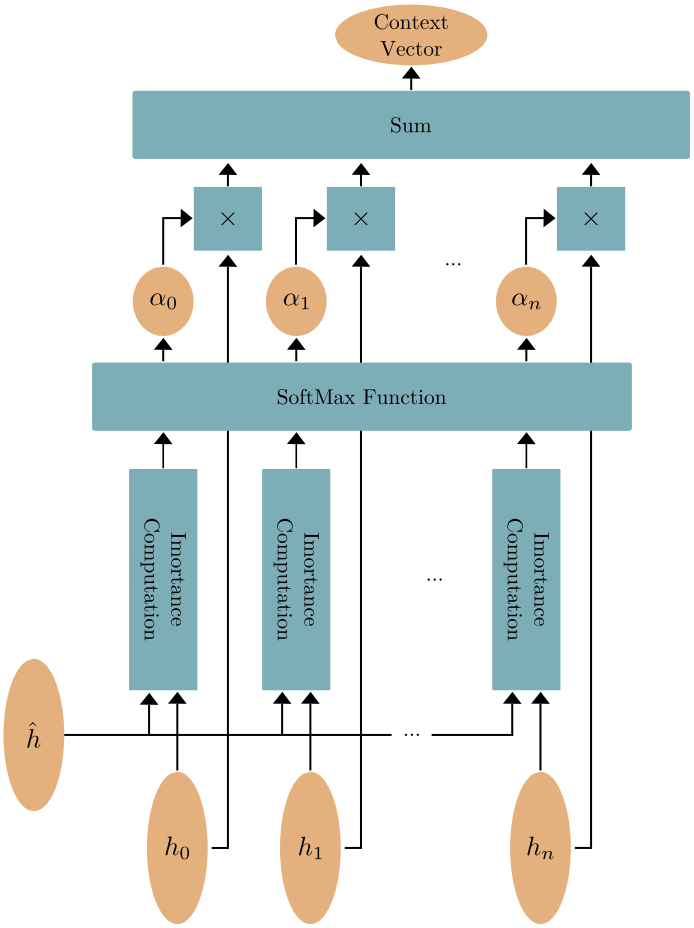

Figure 3: Structure of a computation of the context vector in a recurrent architecture.

of the data produced by the encoder, the hidden states, is discarded, keeping only the last hidden state. As a consequence an arbitrary large amount of data, the input tokens, has to be compressed in a single hidden state, whose size is fixed by the model architecture. The following idea improves over this direction [3]. Figure 3 shows the structure we are going to present.

Let's assume that the encoder has generated the hidden states $h_0, \ldots, h_n$, and let $\hat{h}$ be the current hidden state in the decoder. In the previous architecture we would need a fixed context vector generated from the encoder. What we do now is to generate a context vector dependent on $\hat{h}$. The first step is to have a measure of how important is a given hidden state h_i according to $\hat{h}$. In the most simple version of the architecture, this measure is simply the dot product $h_i^\top \hat{h}$, but other differentiable functions of the two vectors could

be used. Once the importances $x_0, ..., x_n$ are given, they are converted on a distribution probability over the hidden vectors of the encoder. This is done through the SoftMax function:

$$\alpha_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)}.$$

The idea is that the probability distribution measures how relevant is each hidden state for the current prediction, and thus how much attention should be given to each of them. The distribution is in fact called attention. With this in mind, the context vector is simply the mean of the hidden states, weighted through the attention:

$$c = \sum_i \alpha_i h_i.$$

This is very important because attention allows the gradient to "take shortcuts" to get to the first layers without having to pass through all the following ones.

One important way to scale up in size and complexity all the previous models is to have several layer the recurrent architectures. These leads to have a first layer similar to the ones described before: it takes as an input the previous hidden state, the embedding of a token, and in some cases a context vector. Each of the subsequent layers takes in input the previous hidden state for that layer, the hidden state of the previous layer, and sometimes the context vector.

2.1.2 Transformers

The next iteration on these design led to the current State of the Art models like GPT-3.5 or Llama. The idea is to get rid entirely of the recurrent part, focusing only on the attention [26]. The heart of these models is the transformer. It is a layer that takes as an input a set of d-dimensional vectors $e_1, ..., e_n$ and returns a set of vectors of the same size. By means of a multiplication by some fixed matrices W_k, W_q and W_v of size $d \times d_h$ where d_h is called hidden size, the vectors e_i are mapped to three sets of vectors: keys $K_i = W_k e_i$, queries $Q_i = W_q e_i$ and values $V_i = W_v e_i$. The keys perform the same role of the hidden states h_i when choosing the importances in the previous discussion. The queries are the analogous of the vector $\hat{h}$, while the values are the vectors that will be combined through a weighted average.

For each query vector, a measure of importance is computed as before as the dot product with each key value. This result is often scaled by the square

root of the number of dimensions d_h, which while it does not improve the expressiveness of the model, it balance the fact that increasing d_h is reflected in bigger dot products. These importances are then processed in a SoftMax function to give the attention probability distribution:

$$\alpha_i = \text{SoftMax}((Q_i^\top K_0, \ldots, Q_i^\top K_n)/\sqrt{d_h})$$

where

$$\text{SoftMax}(y_1, ..., y_n)_i = \frac{\exp(y_i)}{\sum_j \exp(y_j)}.$$

The output is again obtained by means of a convex combination:

$$O_i = \sum_j \alpha_{i,j} V_j$$

The previous relationships can be expressed in a more compact form seeing the vectors K_i, Q_i and V_i as column of the matrices K, Q and V. Thus we obtain:

$$O = \text{SoftMax}(QK^\top/\sqrt{d_h})V \tag{1}$$

Since the keys, queries and values are obtained from the same set of vectors e_i, the expression 1 is called self-attention head. Figure 4 shows the described architecture.

Usually more than a single self-attention head is present in a transformer layer. In an m-headed transformer there are m triples of matrices $W_k^{(i)}$, $W_q^{(i)}$ and $W_v^{(i)}$, and each of them is used to compute the respective output $O^{(i)}$ for $i = 1, ..., m$. These outputs are concatenated and brought back to the right dimension by means of a linear projection through a matrix W_o of size $md_h \times d$:

$$y = \text{Concat}(O^{(1)}, ..., O^{(m)})W_o.$$

Multiple transformer layers can be stacked on top of each other. Since the only source of non-linearity is the SoftMax function, which appears only in the coefficients of the values, vectors usually go through a Feed Forward Network (FFN) between a transformer layer and the other. These networks are usually very simple, often in the form

$$\text{FFN}(y_i) = W_1\sigma(W_2 y_1 + b)$$

where σ is an activation function, usually a ReLU, and W_1, W_2, b are parameters of suitable size. This component is also called Multi-Layer Perceptron (MLP).

As in most deep learning models, skip connections and layer normalizations are also added.

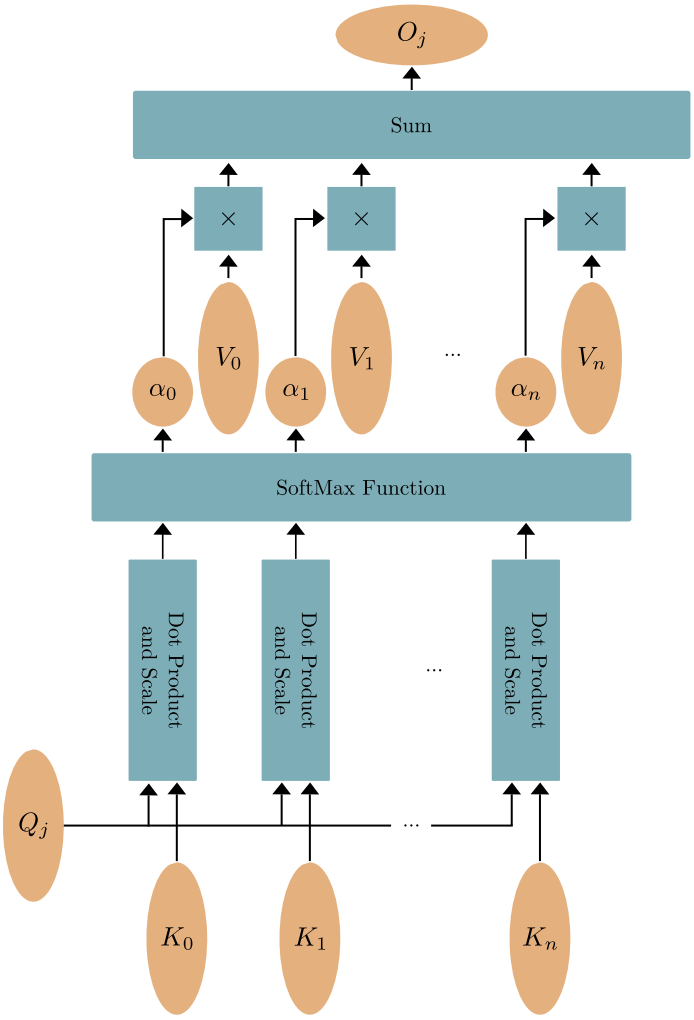

Figure 4: The architecture of a self attention head. This structure is repeated for every index j.

In the recurrent models the order of the tokens was given to the model implicitly, because the their embedding were fed in the right order. The same does not apply in transformer models: a permutation of the inputs leads just to the same permutation on the outputs. In other words the transformer cannot distinguish the order of the inputs by itself, and additional informations are needed. While this can add to the complexity of the model, it also means Transformers are more flexible by recurrent models.

The most common way to add temporal information is performed by adding to each input vector of the transformer x_i a fixed vector dependent

by time only p_i. Thus while x_i changes for each run of the transformer, which will have different inputs, p_i is fixed among all these runs. There are different choices for p_i. One is to have it completely learnable by SGD. Another is to fix it, usually to a vector in the following form:

$$p = \begin{pmatrix} \sin \omega^{1/d} i \\ \cos \omega^{1/d} i \\ \sin \omega^{2/d} i \\ \cos \omega^{2/d} i \\ \vdots \\ \sin \omega^{h/2d} i \\ \cos \omega^{h/2d} i \end{pmatrix}$$

where usually ω is taken to be 10,000.

This kind of vectors has shown to perform well. A possible line of reasoning for its justification is that vectors of sines and cosines can be translated in time with just linear transformations, which are easily performed within the model.

We observe that the length of the path between each output vector and each input embedding is not dependent by the distance between the corresponding token, but they interact directly in the attention computation. This completely solves the biggest issue of the recurrent models.

Another big advantage of transformer based models respect to recurrent ones is that all the computations can be carried out in parallel, instead of having to find the hidden state of the previous iteration in order to be able to start the next one. This allows a better usage of GPUs and therefore smaller training times.

Moreover, in a training phase, all the outputs can be computed in one single sweep. This should be done carefully though, because the prediction of the i-th cannot be performed by having as input the $i+1$-th token, or even the i-th token itself. This can be done by tweaking the attention matrices, imposing that the attention for the i-th query and the j-th key are zero whenever $i \leq j$. The easiest way to accomplish this is by setting the corresponding importances to minus infinity, which in turn will translate to zero through the SoftMax function. This correspond to add to the importance matrices some upper triangular matrices.

Attention masks can be useful for another purpose as well. Until now we considered a single pair input/output going through the model at each step of the training process. However usually deep learning models are trained using minibatches of data for each step. This can be accomplished by running multiple "single pass" steps, one for each element of the minibatch, and

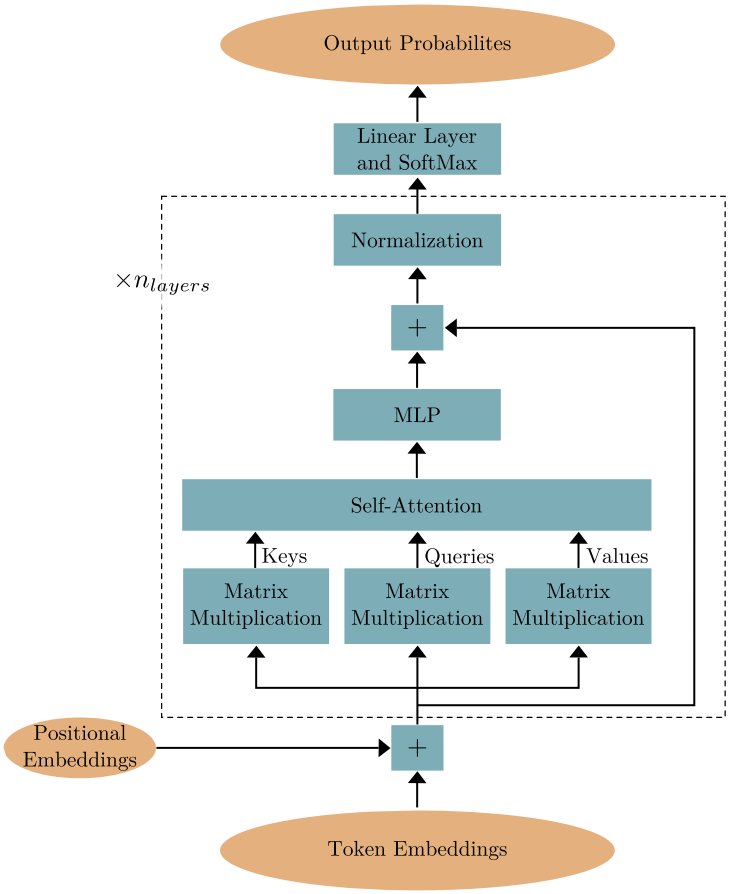

Figure 5: Example of a typical transformer architecture.

accumulating the gradient. It may be more performant to deal with the minibatch in a single pass. This can be done by concatenating all the data for the minibatch in a new dimension of the matrices, and the model doesn't change much to accommodate this modification. However in general the sequence length will not be constant through the minibatch. Thus all the sequences are padded to the max length within the minibatch, and the attention mask is tweaked to avoid that any output depends on a padding token. A typical transformer architecture is shown in Figure 5.

All of this tweaks allows the model to grow in size while still being trainable. In the last year in fact there have been proposed several of such models, varying in size between a few billions parameters to hundred of billions of parameters. These are called Large Language Models, or LLMs. Examples are the GPT family developed by OpenAI, and the Llama family, developed by Meta.

2.1.3 Universality of Transformers

Despite their simplicity, transformers have shown to be a very powerful component in modeling sequential data. In this section we present some results that justify their expressiveness. We start by investigating which class of functions transformers can describe, following the approach of [27].

We define a function from $\mathbb{R}^{d\times n}$ to itself to be a (h, m, r)-transformer if it can be described by a transformer layer with h heads of size m with a MLP that has hidden size r. We assume the transformer to have skip connections, but not layer normalization stages. We then define the set of transformer networks as

$$\mathcal{T}^{h,m,r} = \{g \colon \mathbb{R}^{d\times n} \to \mathbb{R}^{d\times n} \mid g \text{ is a composition of } (h, m, r)\text{-transformers}\}.$$

As we noted in a previous section, transformers without positional encodings have no informations on the order of the input vectors. Therefore the functions described by transformer networks must have this property as well. We then define a function $f \colon \mathbb{R}^{d\times n} \to \mathbb{R}^{d\times n}$ to be permutation invariant if for every permutation matrix $P \in \mathbb{R}^{n\times n}$ and every input $X \in \mathbb{R}d \times n$, f satisfies $f(XP) = f(X)P$. Moreover the transformer network functions are composition of continuous function, and hence continuous themselves.

Let $\mathcal{F}_{\text{PE}}$ be the set of functions from $\mathbb{R}^{d\times n}$ to itself that are continuous, with compact support, and that have the permutation equivariant property. Then the following theorem states that transformer network functions are good approximators of functions in $\mathcal{F}_{\text{PE}}$ [27].

Theorem 1 For every $f \in \mathcal{F}_{\text{PE}}$, $\varepsilon > 0$ and $1 \leq p < \infty$, there exists a transformer network function $g \in \mathcal{T}^{2,1,4}$ such that $d_p(f, g) < \varepsilon$, where d_p is the distance defined as

$$d_p(f, g) = \left(\int \|f - g\|_p^p \right)^{\frac{1}{p}}$$

and $\|\cdot\|_p$ is the term-by-term p-norm.

We can present a similar result for transformer networks with positional encodings. Let us define their set as

$$\mathcal{T}_P^{h,m,r} = \{g \mid \ g(X) = f(X + E) \text{ for some } f \in \mathcal{T}^{h,m,r},\ E \in \mathbb{R}^{d\times n}, \text{ for all } X\ \}$$

Let then $\mathcal{F}_{\text{CD}}$ be the set of continuous functions from $\mathbb{R}^{d\times n}$ to itself with compact domain. In other words $\mathcal{F}_{\text{PE}}$ are the functions in $\mathcal{F}_{\text{CD}}$ that satisfy the permutation equivariant property. The following theorem holds [27]:

Theorem 2 For every $f \in \mathcal{F}_{\text{CD}}$, $\varepsilon > 0$ and $1 \leq p < \infty$, there exists a transformer network with positional encodings $g \in \mathcal{T}_P^{2,1,4}$ such that $d_p(f, g) < \varepsilon$, where d_p is defined as in the previous theorem.

We can also observe an interesting expressiveness of transformer networks under the lens of accepted languages, following the work presented in [23]. We restrict ourselves in working with rational numbers. We define then a seq-to-seq network as a function that take in input a sequence of vectors $X = (x_1, x_2, \ldots, x_n)$, where $x_i \in \mathbb{Q}^d$, a seed vector $s \in \mathbb{Q}^d$, a natural number r and outputs a sequence of r vectors $Y = (y_1, y_2, \ldots, y_r)$, where $y_i \in \mathbb{Q}^d$.

We can see that the transformer we defined in Section 2.1.2 is not a seq-to-seq network because the SoftMax function is not a rational function. In the following discussion we will thus substitute it with the HardMax function, which is the indicator function of the ArgMax of its arguments. The two functions are similar, in a sense that they distribute the attention with a distribution of probability over the value vectors. The output of the seq-to-seq transformer is generated in an autoregressive manner.

We define a language recognizer as a tuple (Σ, f, N, s, F), where Σ is a finite alphabet, $f\colon \Sigma \to \mathbb{Q}^d$ is an embedding function, N is a seq-to-seq network, $s \in \mathbb{Q}^d$ is a seed vector, and $F \subseteq \mathbb{Q}^d$ is a set of final vectors. We say that N accept a string of characters $w \in \Sigma^*$ if exists a natural number r such that $N(f(w), s, r) = (y_1, y_2, \ldots, y_r)$ and $y_r \in F$. To make sure that all the relevant computations happen in the network N and not in the function f or the complexity of F, we add the hypothesis that f should be computed on a Turing machine with linear complexity over the size of Σ, and that the indicator function of F should be computed on a Turing machine with linear complexity over the number of bits of d.

We say that a class of seq-to-seq networks is Turing complete, if for every language L decidable by a Turing machine, there exist a language recognizer with a seq-to-seq network in that class which accept exactly the strings in L.

The main result in [23] is the following:

Theorem 3 The class of transformer networks with positional encodings is Turing complete.

We point out however that the result holds only under the hypothesis of infinite precision, which is unrealistic in architectures of common usage.

2.1.4 The Llama-2 Models

The Llama-2 [24], [25] are a family of LLMs sharing the same underlying architecture and available in different sizes: 7B, 13B, 34B and 70B, where the number counts the number of trainable parameters of the model. The

architecture is the same transformer-based decoder described in the previous section, with the following modifications:

1. The normalization layers, used to prevent internal covariate shifts, are substituted with Root Mean Square layer normalizations (RMSNorm) [28]. The usual normalization layers performs affine transformations on the hidden states such that the activations in a layer have mean zero and standard deviation one. RMSNorm simply scales those activation such that the root mean square is zero. This has shown to perform almost as better as the original version in terms of loss, but is faster to compute.

2. The positional encoding have been replaced by Rotary Position Embeddings. The idea is to add positional information at each layer of the model, modifying the key and query vectors by a suitable function dependent by time. Let this modification be represented by the functions $f_k(y_n, n)$ and $f_q(y_n, n)$, where y_n is the n-th vector of the previous layer or the token embedding multiplied by the matrix W_k or W_q. Let's assume that the hidden dimension to be an even number d, which is reasonable since for performance reasons typical values of dimensions are large powers of two or some of their multiples. Then a good choice could be

$$f_{\{q,k\}}(y_n, n) = R_{\Theta,n} y_m$$

where $R_{\Theta,n}$ is a matrix in the following form:

$$R_{\Theta,n} = \begin{pmatrix} R(n\theta_1) & 0 & \dots & 0 \\ 0 & R(n\theta_2) & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & R(n\theta_{d/2}) \end{pmatrix},$$

$$R(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix},$$

and $\Theta = \{\theta_1, \dots, \theta_{d/2}\}$ with $\theta_i = 10000^{-2(i-1)/d}$.

This choice has a nice property. Let x_n and x_m be the outputs of the previous layer. Then the attention importance relative to the n-th query and m-th key using the Rotary Embedding can be written as

$$\begin{aligned} I_{n,m} &= (R_{\Theta,n} W_q x_n)^\top (R_{\Theta,m} W_k x_m) \\ &= x_n^\top W_q^\top R_{\Theta,m-n} W_k x_m. \end{aligned} \tag{2}$$

The matrix $R_{\Theta,m-n}$ is an orthogonal matrix, which helps the flowing of the gradient during training. Moreover from equation 2 we can see that this is a kind of relative encoding, which, in contrast with the absolute encoding, add informations only about the distance of two tokens, and not their position in the whole sentence. For many tasks this is a desirable property. One final remark is that the matrices $R_(Theta, n)$ are very sparse, and thus the computation of the products can be performed in a $\mathcal{O}(d)$ complexity.

3. The MLP layers has been replaced with a Swish Gated Linear Unit (SwiGLU). The function Swish is defined as

$$\text{Swish}(x) = x\sigma(x)$$

where σ is the sigmoid function. Given a hidden dimension of the model d_h and a new dimension d_{MLP} for the MLP, we define the Gated Linear Unit as be parametrized by the W_{gate}, W_{up} and W_{down}, of size $d_h \times d_{\text{MLP}}$ for the first two, and $d_{\text{MLP}} \times d_h$ for the last one. Let's denote the component-wise multiplication of two vectors with the symbol $\odot$. Then the layer is defined as

$$\text{MLP}(x) = W_{\text{down}}(\text{Swish}(W_{\text{gate}}x) \odot W_{\text{up}}x)$$

This approach has been proven to have a better behaviour than a plain MLP layer.

2.2 Generative AI

Generative AI is a branch of Machine Learning whose objective is to sample new data from an unknown distribution, given a dataset of independent samples from that distribution.

One possible approach that have been used is the one of Generative Adversarial Networks (GANs)[10]. The idea is to have a generator neural networks that generates synthetic samples from white noise, and a discriminator networks that should be able to tell synthetic samples from original ones. The two networks are then trained together, with the discriminator having the objective of making accurate prediction over the origin of its input, and the generator having the objective of "being able to fool" the discriminator. In other words, the discriminator weights are updated in such a way that try to minimize the loss associated to the classification task, while the weights of the generator are updated in such a way that try maximize that loss.

Some problems however arise from training GANs. The first one is that the convergence of the joint training of the two networks is often hard. Some

steps have been made to tackle this problem. WGANs [2] are one possible improvement, for example. Another problem is that often generators end up sampling only a partial region of the whole distribution, the one that creates more difficulties to the discriminator.

Another approach to generative AI, that has been proved very successful in the last years with popular models such as DALL-E [4], is the one that uses variational autoencoders [14].

These models are again split in two parts, an encoder and a decoder. The decoder maps the samples in the dataset to vectors in a latent space, while the decoder tries to reconstruct the original data from the latent vector. The training has the objective of minimizing the reconstruction error while keeping the distribution of projections onto the latent space similar to white noise. After the training is complete, the generation is done by just sampling a latent vector from actual white noise and computing the sample through the decoder.

Diffusion based models for generating images like DALL-E or Midjourney have an architecture very similar to variational autoencoders where the latent space has the same dimensionality of the sample data and the encoder is a function that adds random noise to an image.

2.3 Explainable AI

New AI based on deep-learning technologies show outstanding State-of-the-Art performances in many fields. However, most of models based on deep-learning, random forests, or SVM are opaque, which means that the inner reasoning that brought to the high accuracy results are not easy to extract from the prediction. Those models are then also called black-box models. Thus the last years witnessed to an increasing interest in the field of eXplainable Artificial Intelligence (XAI), whose objective is to augment the predictions with explanations understandable by humans.

Following the taxonomy of [5], we classify the most common XAI techniques as follows:

- Some models are *explainable by design* or intrinsically explainable, which means that their reasoning behind each predictions is transparently available to the user. Examples are decision trees or linear regression models.

- *Post-hoc explanations* are those which complement already trained opaque black-box models, which are the source of the predictions. The explanations are made using the outputs or other internal states of

the black box. Examples are the methods that work in tandem with deep-learning models.

Focusing on the post-hoc methods, we categorize explanation algorithms on two distinct axis. The first one describes what is being explained:

- *Local* explainers on the other hand try to explain only a single prediction. They try to capture the logic of the black-box only in a neighborhood of the instance to be explained.
- *Global* explainers try to explain the whole logic within the black-box model. The explanation is thus meaningful for each instance of the data.

The second axis focus on the kind of information the explainer leverages on:

- *Model-Specific* methods can use some data obtained from within the associated prediction model. Examples are the gradient from deep-learning models or attention values from transformer models.
- *Model-Agnostic* methods can only use the predictions given by the black-box. This means that any kind of black-box can be successfully used in tandem with this kind of explainers.

There are different types of methods to obtain explanations. Some are better suited to specific data shapes, while others are more universal. Examples of explanation methods that can be used with any prediction task are prototypes and counterfactuals. Some methods like rule-based explanations or feature importances are used mostly on tabular data. Images are often explained using saliency maps or concept attribution, while text data uses often sentence highlighting or attention based methods [5].

We proceed to show some important groups of explanation techniques.

- *Prototypes*. The user is presented with data which may taken from a dataset or generated artificially, that gives examples similar to the input of the predictions that get a similar prediction.
- *Counterfactuals*. This method is similar to the prototypes, but work in the opposite direction: the counterfactuals are data similar to the input but with different prediction.

- *Feature importance.* Let us assume that each instance of the data to be explained is a sequence of features $(x_1, x_2, \ldots, x_n)$. The explanation consists in a vector $e = (e_1, e_2, \ldots, e_n)$ where each component e_i gives the importance of the corresponding feature x_i. Usually e positive value of e_i can be interpreted as the feature x_i contributing positively toward the given prediction, while negative values indicate that the feature contributes negatively.

- *Rule based.* The goal of this method is to find a set of rules that mimic the black-box at least on a subset of the data near the input to be explained. A rule consists in a boolean condition over the data and a prediction to be assigned when the condition is satisfied.

2.3.1 Decision Tree Classifiers

An important example of intrinsically explainable models are the tree classifiers [11], [6]. Let $\mathcal{D} = (x^{(1)}, x^{(2)}, \ldots, x^{(k)})$ be a dataset of n-dimensional vectors, and let $(y_1, y_2, \ldots, y_k)$ be a sequence of labels from a finite set $\mathcal{C}$. A decision tree classifier is a description of iterative splits of the n-dimensional space in partitions that contains data points x_j with similar labels.

A decision tree classifier is a binary tree whose nodes are represented by a couple i, t where i is the index of a feature, so $i = 1, \ldots, n$, and $t \in \mathbb{R}$ is a threshold value. A subset of the dataset is associated to each node of the tree: the whole dataset is associated to the root node, and if the subset D is associated to a non-leaf node $\theta = (i, t)$ then the subsets

$$D_{\text{left}} = \{x \in D \mid x_i \leq t\}$$
$$D_{\text{right}} = \{x \in D \mid x_i > t\}$$

are associated to its left and right child respectively. We finally associate a label to each leaf of the tree. Thus to each input x the tree classifier predicts the label of the leaf associated to x through the rules.

One of the most common ways of finding a good decision tree is the CART algorithm [11]. We first associate a number that measures the quality of the split in each inner node.

$$G(\theta) = \frac{|D_{\text{left}}|}{|D|} H(D_{\text{left}}) + \frac{|D_{\text{right}}|}{|D|} H(D_{\text{right}})$$

where the function $H(D)$ returns the impurity of the set D. There are several options to choose this function from, the most popular is the Gini

index computed as

$$H(D) = \sum_{j=1}^{n} p_j(D)\,(1 - p_j(D))$$

where

$$p_j(D) = \frac{1}{|D|} \sum_{(x,y)\in D} I(y = j)$$

and I is the indicator function.

The tree is then built recursively and greedily, by assigning to each node the rule θ that minimizes $G(\theta)$, until a maximum depth is reached. In a second phase of the algorithm, the tree is pruned by removing nodes to reduce the complexity of the predictor.

To each non-leaf node we can associate an impurity reduction measure defined as

$$\mathrm{Red}(\theta) = \frac{|D|}{|\mathcal{D}|}\left(H(D) - \frac{|D_{\mathrm{left}}|}{|D|}H(D_{\mathrm{left}}) - \frac{|D_{\mathrm{right}}|}{|D|}H(D_{\mathrm{right}})\right).$$

This measure describes how successful each node is in decreasing the impurity in a region. We can use this impurity reduction to measure the impact of a feature on the decisions taken by the classifier. Given a decision tree T and a feature j, its importance $\mathrm{FI}(T, j)$ is the sum of the impurity reduction over all inner nodes that split on the feature j. More formally if

$$\Theta_j = \{(i, t) \in T \mid i = j\}$$

then

$$\mathrm{FI}(T, j) = \sum_{\theta\in\Theta_j} \mathrm{Red}(\theta).$$

We will use the feature importances given by a tree classifier to investigate whether a local explainer based on trees match the neural activation within the black box.

3 Problem Description

The main task of this thesis is to make predictions augmented with an explanation over medical data, and try to use the explanations to gather some insight over the prediction methods.

The data we will work on is a dataset of patients. Each patients is modeled as a sequence of visits, and at each visit they are assigned a certain number of diagnostic codes. The codes are organized hierarchically, in a structure called ontology. The data is formally described in section 3.1.

The goal of the prediction is to tell, given a patient history of visits, which codes will occur in the next visit. To do so, we will build a black-box predictor, that we will also call predictive model. The modeling of the predictor is done in section 4.1.

In addition we will give an explanation to the prediction with a post-hoc local method. We will follow the approach of [20], generating a dataset of data points similar to the patient to be explained, and use it to fit a decision tree classifier. From the tree, the explanation can be extracted in the form of decision rules. The description of the explanation pipeline can be found in section 3.3.

We will expand the existing work by building a predictive model (section 4.1) that outperforms the original one in the accuracy of the prediction, while being still not at the same level of the state of the art models. The strength of our solution however is that it uses uniform transformer layers that leverage only the diagnostic codes and not the whole ontology, in contrast with other models (for example [22]). In particular our model assigns an attention matrix to each input code in a clear way. This allows us to extract a measure of relevance over the possible codes according to the predictor. We will study this measure comparing it with a feature importance measure extracted from the decision trees (section 4.3).

We will also propose a new method to generate a neighborhood for building the explanation (section 4.2).

3.1 Data Type

Let $\mathcal{C}$ be a finite set of discrete elements called codes. A visit is defined as any subset of $\mathcal{C}$. A patient is defined as a finite ordered sequence of visits. A code is allowed to appear more than once within different visits of the same patient. Let $\mathcal{V}$ be the set of all possible visits, and $\mathcal{P}$ be the set of all possible patient.

A dataset $\mathcal{D}$ consists of a sequence of patients from $\mathcal{P}$. We assume that the patients in $\mathcal{D}$ are sampled independently from an underlying and unknown

distribution μ.

In many fields, and in particular the medical one, which is the one of our interest, some categorical data is not just made of indistinguishable labels, but rather by leaf-nodes of a tree. In other words, there exists a hierarchical structure that groups together codes at different layers of the tree.

This can be formally described as follows. Let $\mathcal{C}_{\text{in}}$ be a set of "inner nodes", and let $\mathcal{C}' = \mathcal{C} \cup \mathcal{C}_{\text{in}}$. Let $\mathcal{T}$ be a tree with nodes $\mathcal{C}'$ and whose leaves are all and only the codes $\mathcal{C}$. This tree is called ontology and gives some information over the codes in $\mathcal{C}$ that appears in the dataset. We will assume that codes that are closer with respect to the distance on the graph $\mathcal{T}$ to be more similar than codes that are less close.

3.1.1 Ontological Distance

As done in [20] we use the ontology to define a distance on the possible patients $\mathcal{P}$. To do that, we will use a distance over the codes $\mathcal{C}$ and the visits $\mathcal{V}$.

We start describing a code-to-code distance. It is the Wu-Palmer similarity score, which is one of the most commonly used for the ICD ontology. Let c_1 and c_2 be two codes in $\mathcal{C}$. Let L be their lowest common ancestor on the tree $\mathcal{T}$, and let R be the root of $\mathcal{T}$. Let $d(c_1', c_2')$ be the distance between codes in $\mathcal{C}'$ which measures the smallest number of steps needed to reach c_2' starting from c_1' and moving along the edges of the graph $\mathcal{T}$. The Wu-Palmer similarity score is then defined as

$$\text{WuP}(c_1, c_2) = \frac{2d(L, R)}{d(c_1, L) + d(c_2, L) + 2d(L, R)}$$

We observe that $0 \leq \text{WuP}(c_1, c_2) \leq 1$ for each pair of codes, and the minimum value 0 is reached when L is the root of the tree, while the maximum value 1 is reached when $c_1 = c_2 = L$.

We can then define a visit-to-visit distance. The approach of [20] is to use an edit distance weighted through the Wu-Palmer similarity. However to do so, we would need to choose an order between the codes of each visit, while the codes do not have a natural order. For this reason we will follow a different approach, substituting the edit distance with a measure invariant under reordering of codes. Let $V_A = \{c_1^A, \ldots, c_a^A\}$ and $V_B = \{c_1^B, \ldots, c_b^B\}$ be two visits composed by a and b codes respectively. For each code in V_A we choose the best similar code in V_B. We them sum all of the distances between the best pairs to get an asymmetric distance:

$$d_{\text{asym}}^{\mathcal{V}}(V_A, V_B) = \sum_{i=1}^{a} \min_{j=1\ldots b} \text{WuP}(v_i^A, v_j^B).$$

We can symmetrize the above expression taking the maximum of the two permutations:

$$d^{\mathcal{V}}(V_A, V_B) = \max(d^{\mathcal{V}}_{\text{asym}}(V_A, V_B), d^{\mathcal{V}}_{\text{asym}}(V_B, V_A)).$$

Assuming to have a precomputed table with all the distance pairs $\text{WuP}(c_1, c_2)$, the computation of $d^{\mathcal{V}}(V_A, V_B)$ has a complexity of $\mathcal{O}(n^2)$, where n is an upper limit on the size of the visits.

Finally we are ready to describe a patient-to-patient distance $d^{\mathcal{P}}$. We do that through the Dynamic Time Warp (DTW) Algorithm. This algorithm gives a measure of similarity between two time series that can differ in speed. The idea is to find associations between two elements of each series, in our context they are visits of two patients, subject to the following constraints:

- Every element of the first sequence must be associated to an element of the first one;
- The first and the last elements of each sequence must be associated between them, but they not to be their only association.
- The associations must be monotonical: Let the i-th element of the first sequence is associated to the j-th element of the second one, and a similar things happens for the i'-th element in the first sequence and the j'-th element of the second one. Then $i < i'$ implies $j \leq j'$.

We can give a cost to each way of associating two series: it is the sum of the visit-to-visit distances between each associated pair. The DTW similarity is then defined as the minimum cost between all the associations that respect the previous conditions.

This optimization problem can be solved with a dynamic programming approach. The algorithm to compute the DTW is Algorithm 1.

It should be noted that the measure we have defined is not a distance in the sense of the metric spaces, in fact it cannot guarantee the triangular inequality. Moreover if the number of visit per patient is bounded by m and the number of codes per visit is bounded by n, then there will be m^2 iterations within the DTW algorithm, each of those will require a computation of the visit-to-visit distance, which costs $\mathcal{O}(n^2)$ iterations. Thus the total cost of the algorithm is $\mathcal{O}(m^2 n^2)$.

3.2 Predictor Models

We model a predictor as follows: A predictor model $h(\cdot, \theta)$ parameterized by a vector of parameters $\theta \in \mathbb{R}^d$, for some number of dimensions d, is a

Algorithm 1 DTW Algorithm

Require: s: array of length n
Require: t: array of length m
Require: d: distance between elements of s and elements of t
  function DTW(s, t) ▷ Computes the DTW distance between s and t
    $A \leftarrow$ array $[0 \dots n, 0 \dots m]$
    for $i \leftarrow 0 \dots n$ **do**
      for $j \leftarrow 0 \dots m$ **do**
        $A[i, j] \leftarrow +\infty$
      end for
    end for
    $A[0, 0] \leftarrow 0$

    for $i \leftarrow 1 \dots n$ **do**
      for $j \leftarrow 1 \dots m$ **do**
        $c \leftarrow d(s[i], t[j])$
        $A[i, j] \leftarrow c + \min(A[i-1, j], A[i, j-1], A(i-1, j-1))$
      end for
    end for
    return $A[n, m]$
  end function

function

$$h(\cdot, \theta) : \mathcal{P} \longrightarrow [0, 1]^{|\mathcal{C}|}.$$

The underlying idea is for h to be a predictor of the next visit for a given patient. In fact we can see $h(p, \theta)$ as a distribution on the visits $\mathcal{V}$, where we assume that each code is chosen independently from each other, and the i-th code has a probability of appearing given by the i-th component of $h(p, \theta)$. For example the vector $x = (x_1, \dots, x_{|\mathcal{C}|})$ assigns to the visit $v = (c_{i_1}, \dots, c_{i_n})$ the probability $\prod_j x_{i_j}$. In the following discussion we will identify the vector x with its distribution on $\mathcal{V}$.

Given μ the probability distribution of each patient in the dataset, we can factor it over each visit in the following way: let $p = (v_1, \dots, v_k)$ be a patient. Then the decomposition is given by

$$\mu(p) = \mu_0(v_1)\mu_{\text{next}}(v_2|(v_1))\mu_{\text{next}}(v_3|(v_1, v_2)) \cdots \mu_{\text{next}}(v_k|(v_1, \dots, v_{k-1}))$$

where μ_0 is a probability distribution over the initial visits $\mathcal{V}$, and μ_{next} is a probability distribution over the next visit given the preceding ones. In other words $\mu_{\text{next}}(v|p')$ is the probability of v being the next observed visit after seeing all the visits in p'.

The objective of a training process is to find an optimal parameter $\hat{\theta}$ for which the function $h(\cdot, \hat{\theta})$ is the best approximation of the distribution μ_{next}. Since the latter is never known, we will use the empirical distribution found in the dataset $\mathcal{D}$ as its proxy.

Examples of predictive models are doctorAI [7], which uses a recurrent architecture, and SETOR [22], which uses a transformer architecture that incorporates informations from the ontology.

3.2.1 Evaluation Metrics

Predictor models are typically evaluated using a "Recall@k" metric [22], [20], where k is an integer value. The output of the predictor is a vector of size $|\mathcal{C}|$. To compute the Recall@k, we discretize the output vector assigning value 1 to all the codes whose entry is among the biggest k entries in the vector, and a value of 0 to all the others. We call a predicted code a true positive if it is present in the target visit to be predicted, and has assigned a value of 1 in the discretized prediction. Then Then

$$\text{Recall@k} = \frac{\text{number of true positives}}{\text{number of codes in the target visit}}.$$

Since the recall metrics are not differentiable we will use the average of binary cross-entropy among all the codes as a loss function for training deep learning models, as done in [22].

3.3 Explanation Method

In this section we will present the explanation method used by DoctorXAI.

DoctorXAI [20] is a tool to give prediction with an explanation on medical data composed by two distinct parts. A predictor, called doctorAI [7], and a novel explanation pipeline. The explanation follows a post-hoc agnostic method, and explains locally a prediction by selecting a neighborhood of patients similar to the instance to be explained, and then fitting an inherently explainable predictor such as a decision tree classifier on that neighborhood whose goal is to imitate the black-box behavior. The process is shown in figure 6.

More specifically in DoctorXAI, given a patient history p, a neighborhood of patients similar to p is selected from the training dataset. Then we augment it by perturbing its elements to create a more dense neighborhood. We associate to each element of the neighborhood a label given by the black-box model, and we encode every patient into a tabular form, to create a dataset N. Finally an intrinsically explainable predictor, a decision tree classifier, is

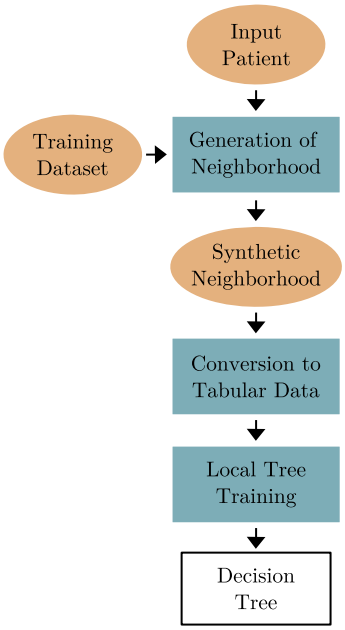

Figure 6: General pipeline used to build an explanation. Given a patient whose prediction for the next visit is to be explained, we show the process to build a decision tree from which to extract an explanation.

trained on N. The final explanation is extracted from this latter predictor, in form of a set of rules. The pipeline to build an explanation is shown in Figure 7. An example of prediction is shown in Figure 8.

We notice that in contrast with other works such as [19], we do not stratify the patients on their attributes like sex or ethnicity.

Let $\mathcal{D}$ be a dataset of real patients for explanation purposes. Let d be the patient-to-patient ontology-aware distance described in section 3.1.1. We then extract from $\mathcal{D}$ the k patients that are the closest to p according to d. This will form the real neighborhood N_{real} of p. The natural number k is a parameter of the algorithm.

The following step is to augment N_{real} with synthetic patients. In order to not introduce too much noise into the final dataset N, synthetic patients are generated only through deletion from real patients. The deletion is performed in an ontology-aware manner, as follows: given a patient $p' \in N_{\text{real}}$, each code of its visits has a fixed probability $\theta \in [0, 1]$ to be deleted. In addition let $D = \{c_1, c_2, \ldots, c_r\}$ be the set of such codes chosen for deletion. Let c'_i be the parent of c_i according to the ontology tree, for each $i = 1, 2, \ldots, r$, and let C' be the sets of the c'_i codes. Then the generated synthetic patient p' is equal to p when all the codes whose parents appear in C' have been deleted. We notice that for construction all the codes in D are deleted. In other words if

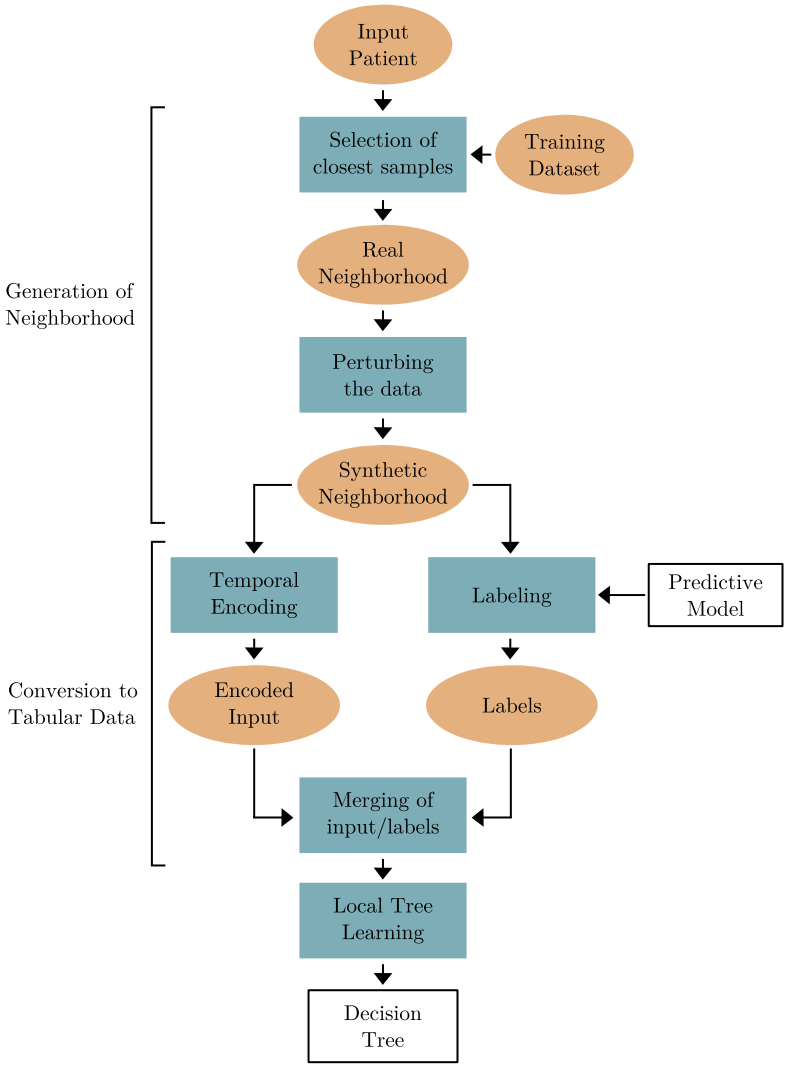

Figure 7: Pipeline used to build an explanation.

a code is deleted, all the children of its parent are deleted too.

We then need to associate a next visit prediction to each element of the neighborhood. This is done by just running the black-box predictor on the data. A note on how the output is extracted should be done. In fact the actual output of the black box predictor is a real number between zero and one for each code, which defines the probability of that code to appear in the next visit according to the predictor. Since these kind of predictors are evaluated with recall at k_{rec} metrics, the predicted visit contains the j-th code when it is one of the first k_{rec} most probable codes.

Since we are interested in explaining a specific prediction of a visit v,

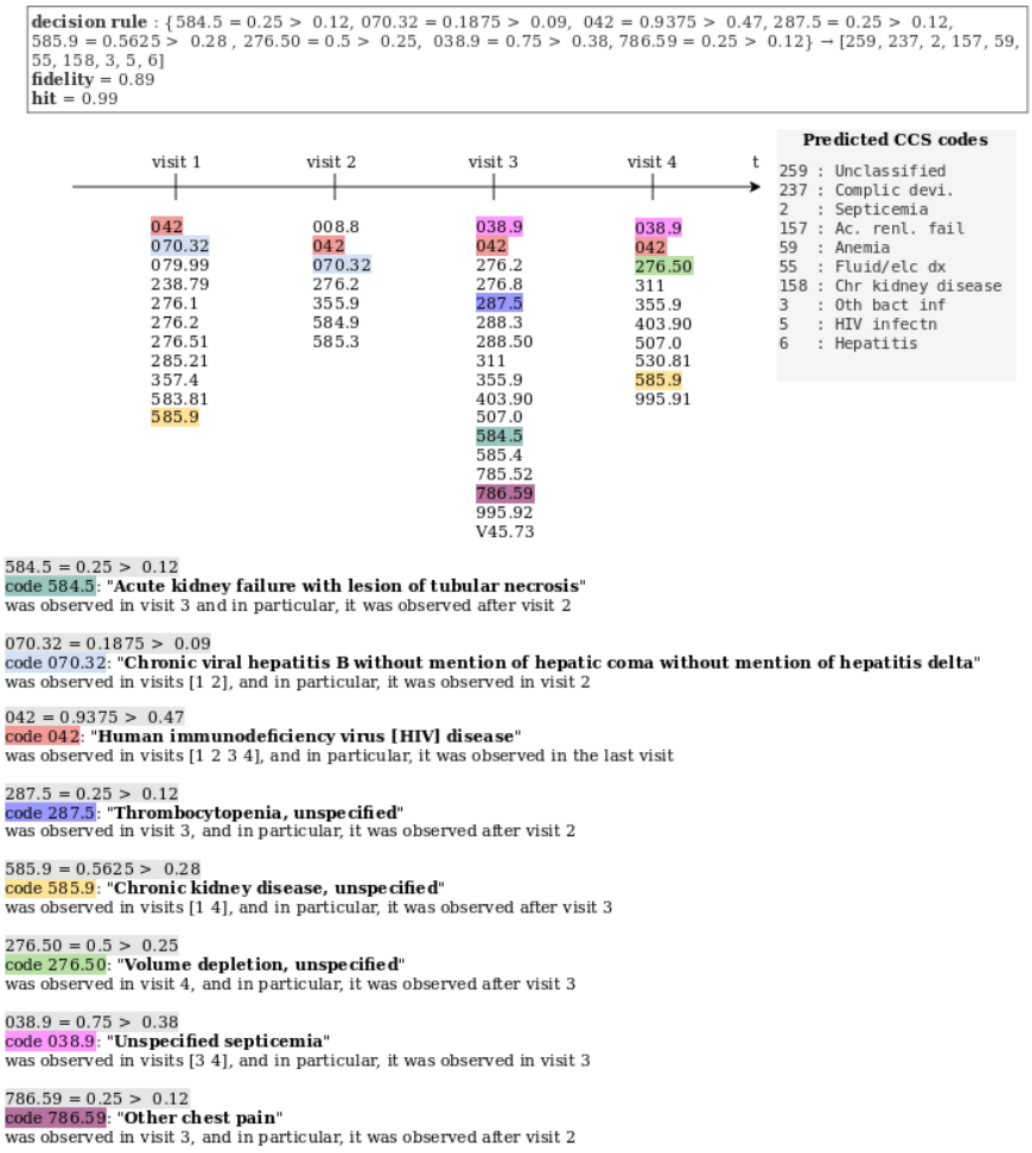

Figure 8: Example of an explanation generated by doctorXAI. The image is taken from [20].

which thus contains exactly k_{rec} codes, we restrict the target labels through the whole dataset N to those codes. Therefore the target output has k_{rec} dimensions.

We then have a dataset consisting in couples of patient history and prediction. The goal is to fit a decision tree classifier on them. Trees however work only on tabular data, while the patients are sequences of codes of variable length. So we need one more step to convert the dataset in a more

suitable form for the decision trees.

The proposed solution is to encode all the codes in a single vector. Let $p = (v_1, v_2, \ldots, v_n)$ be a patient to be encoded. Each visit can be encoded in a fixed-size vector through a one hot encoding. If $\mathcal{C}$ is the set of all possible codes, the vector will have $|\mathcal{C}|$ dimensions. Let $\hat{v_i}$ be the codification of the visit v_i. Then

$$(\hat{v_i})_j = \begin{cases} 1 & \text{if the } j\text{-th code appears in } \hat{v_i} \\ 0 & \text{otherwise.} \end{cases}$$

Let $\lambda \in [0, 1]$ be a fixed discount factor, parameter of the algorithm. In [20] a value of $\lambda = 1/2$ is used. Then we can aggregate the vectors $\hat{v_i}$ with an exponential decay governed by the λ factor. We thus obtain the final encoding

$$\hat{p} = \sum_{i=1}^{r} \lambda^{r-i+1} v_i.$$

A tree is then fitted on this data, the inputs being $|\mathcal{C}|$-dimensional real vectors and the outputs being k_{rec}-dimensional vectors of zeros and ones. The explanation consists in the branches taken by the patient to be explained p when classifying it according to the trained tree. We can show the explanation to a user by highlighting which codes are present in the patient to be explained and correspond to a split node in the tree, as shown in Figure 8.

3.3.1 Evaluation Metrics

Explanations are evaluated with three metrics:

1. Fidelity to the black box, which describes the performances of the tree predictions using the black-box as the ground-truth. This metric is evaluated on a held-out set taken from the neighborhood N. Since this is a multi-label classification task [29], a F_1 score with micro averaging.

2. A hit score, which measures how often the tree and the black-box models agree on the patient to be explained. This is computed as the Hamming-distance between the two outputs.

3. A measure of complexity of the explanation. This is the depth of the leaf in the decision tree corresponding to the patient to be explained, and represents the length of the explanation. This is important because longer rules may be difficult to interpret by humans [15].

We can observe that there is a trade-off to be done between the first two metrics and the third. In fact deeper trees can lead to better agreements between the black-box and the explainable classifier, but will lead to longer explanations.

4 Implementation of the Solution

4.1 Black-Box Model

We build a new black-box model based on transformer architecture modifying the Llama LLM [24], [25]. We observe that the shape of the data of a LLM is simpler than the one we are dealing with. In fact text is defined in this context as a pure sequence of tokens, while our medical data is a sequence of sets of labels, as described in section 3.1. The structure of our model is shown in Figure 9. The main differences with the Llama models are the following:

- The model takes an additional input that codifies the assignment of each code of the patient to its visit. This data is used while computing the positional encodings.
- The attention mask is modified in such a way that allows queries and keys of codes in the same visits to interact with each other regardless of their relative positioning.
- Instead of a linear layer, an attention pooling head is used to extract the output from the last transformer layer.
- We use standard normalization layers instead of Root Mean Square normalization layers.

We will take advantage of the flexibility of the positional embeddings to pass the structure of the data to the model. Our goal is to have an architecture invariant under permutation of codes within the same visit but sensible to permutations of codes from different visit. To do so we will tweak the positional embeddings and the attention masking components.

Let $p = (v_1, v_2, \ldots, v_n)$ be a patient, where $v_i = \{c_1^{(i)}, c_2^{(i)}, \ldots, c_{m_i}^{(i)}\}$ are visits for $i = 1, 2, \ldots, n$. Let p' be the flattened sequence of codes in p, in which the order within each visit is arbitrarily chosen:

$$p' = (c_1^{(1)}, c_2^{(1)}, \ldots, c_{m_1}^{(1)}, c_1^{(2)}, \ldots, c_{m_2}^{(2)}, c_1^{(3)}, \ldots, c_{m_n}^{(n)}).$$

Then let q be the sequence of natural numbers that specify the positional encoding, where q_j is equal to r if p'_j is a code corresponding to the r-th visit in p. We can then feed to the transformer model the vector p' to look up for the corresponding embeddings, one for each code, and the vector q to set the correct positional embedding. Since the positional embeddings are the only part of the transformer where positions matters, the choice of the order of the codes when forming the vector p' does not affect the final result.

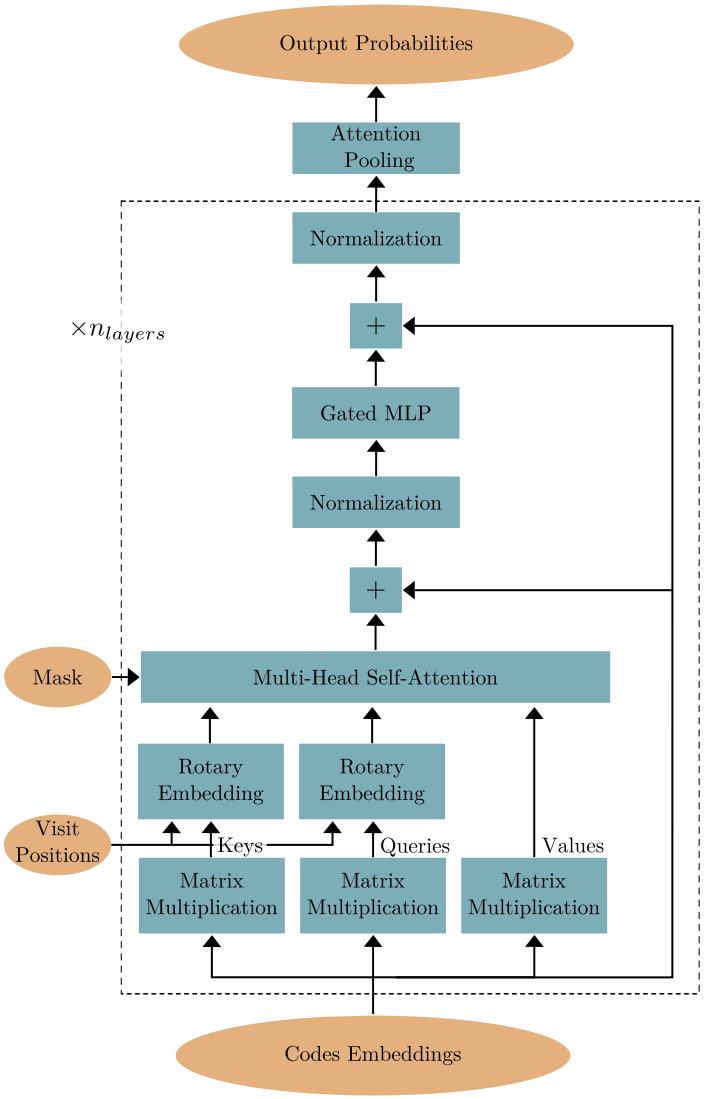

Figure 9: Architecture of the black box model.

We notice that this could not be performed in a recurrent architecture, since permutations in the order of the codes necessarily lead to different results.

Together with the tweaked positional embeddings a modification to the decoder mask should be done. In fact keys and queries associated to codes within the same visit should be always allowed to interact with each other. The mask should still prohibit codes in the future visits to "look into" codes of previous visits. Thus the mask M must have a block-wise triangular form,

as follows:

$$M_{i,j} = \begin{cases} 0 & \text{when } q_i \leq q_j \\ -\infty & \text{otherwise.} \end{cases}$$

At this stage the model is able to correctly handle the input data and its structure. However transformers do not change the shape of the data, but while the input of the model has a length equal to the number of the codes of the patient, the output should have a length equal to the number of the visits. Since in general in the final layer of the transformer there will be a vector for each code in p', we adopt an additional layer to aggregate all the vectors regarding codes of the same visit into a single one. This is called an attention pooling layer. We present two different versions of it. We will treat having the parametrized head or the non parametrized one as a hyperparameter of the model.

The simplest one performs a simple component wise average over the vectors to be aggregated. This keeps the same "consideration" for each code in the visit, and has the nice properties to be differentiable (which is needed for SGD), invariant under the permutations of the codes, and the resulting vector does not have bigger magnitude for longer visits, as a sum would lead to.

The second version is parametrized, and allows for different codes to contribute in different amounts to the visit vector. Its architecture is shown in Figure 10. Let O be the output of the transformer, so it is a $m \times h$ matrix, where m is the number of the codes in the visit $m = \sum_i m_i$, and h is the hidden dimension. We pass O trough an additional transformer layer to obtain a sort of "importance" matrix X. The transformer layer is made up as usual by a multi-self-attention head and a MLP layer. Since the objective of this layer is to pool together the codes in the same visit, the positional embeddings are disabled, and the mask allows keys and queries vectors to interact only if they are relative to codes within the same visit. More formally, the mask M is defined as

$$M_{i,j} = \begin{cases} 0 & \text{when } q_i = q_j \\ -\infty & \text{otherwise.} \end{cases}$$

We now have to act on X such that we can define a probability vector over each visit. We do this separately for each of the h dimensions of the hidden vectors. As usual this conversion is done with a SoftMax function. Let Y the output of this operation and j a selected column, $j \in \{1, 2, \dots h\}$. The matrix Y will have the same shape of X. Let i a selected row index, $i \in \{1, 2, \dots m\}$, such that the it corresponds to the k-th visit. Let a and b

the index of the first and last codes respectively in that visit. This means that

$$a = \sum_{r<k} m_r + 1$$

and

$$b = \sum_{r \leq k} m_r.$$

Then we define

$$Y_{[a:b],j} = \text{SoftMax}(X_{[a:b],j}),$$

where $Y_{[a:b],j}$ denotes the vector such that $(Y_{[a:b],j})_l = Y_{a+l-1,j}$ for $l = 1, ..., b-1$.

The resulting Matrix of the pooling is computed by summing all the vectors in $X \odot Y$ corresponding to each visit.

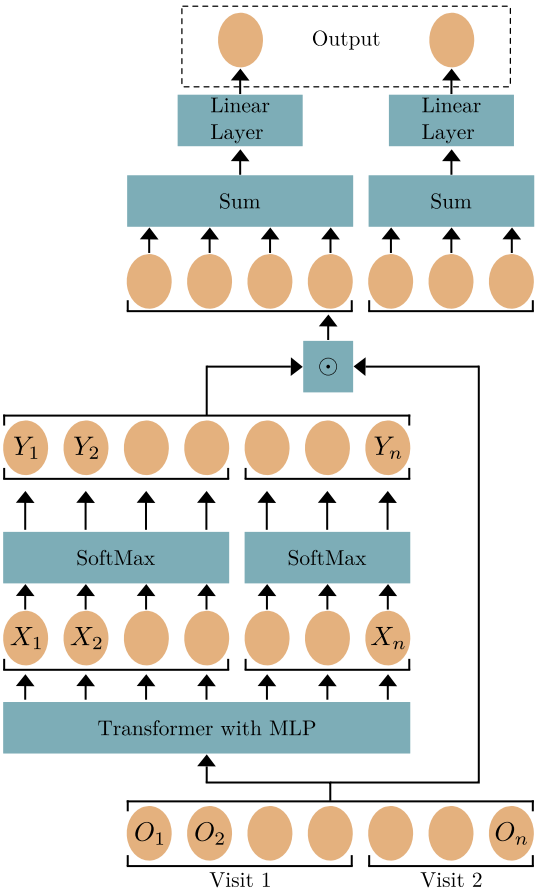

Figure 10: Architecture of the Attention Pooling with parameters. This is an example with two visits, but it can be generalized to any number of visits.

With both the versions of the pooling head, the output is a matrix of shape $n \times h$ where n is the number of visits. A simple linear layer will be enough to bring the output at a shape equals to $n \times |\mathcal{C}|$ that represent the logits for the presence of each codes in the prediction.

Of course only the last vector will be used for inference, but the whole matrix is needed for the training.

Another modification over the original Llama model is that we use the standard normalization layer instead of RMSNorm. We make this choice because training times are not very long with the amount of data we have.

4.2 Generative Explanation Pipeline

Our second proposal is the generation of the synthetic neighborhood for the explanation pipeline described in Section 3.3 (Figure 6). We start with a real neighborhood N_{real} of patients selected to be as close as possible to the patient to be explained p. The notion of "closeness" is again formalized by the ontology-aware visit-to-visit distance of section 3.1.1 (Algorithm 1). We can also perform generative perturbations on a dataset of synthetic patients generated with ontological perturbations.

The ontology pipeline modifies the real patients only by deletion, to avoid introducing noise into the neighborhood. That would mean having to run the black-box model on patients out of the original distribution of the data. Our goal is to find a way to explore the neighborhood of p by augmenting the real neighbors with new codes that are plausible. The idea is to have a model that codifies the distribution of the data, and let the generative process be guided by that model.

Following ideas of diffusion based models presented in Section 2.2, we generate new patients by training a model to convert noise in coherent patients. To do so, we build a model that is able to reconstruct the original patients when some noise is added. Given a real patient p_r, the noise is added by masking some randomly chosen codes present in their visit with a special code called `hole`. The input has thus the same shape as described in section 4.1, but the codes vector is able to contain the new special code.

The model is then composed by the following stack of layers:

1. An embedding layer that associates each code to a h-dimensional vector. It can accept $|\mathcal{C}| + 1$ distinct codes.

2. Several transformer layers. These are built in the same way as in section 4.1, with all the care needed to encode the correct structure of the data through the positional embeddings and the decoder masks.

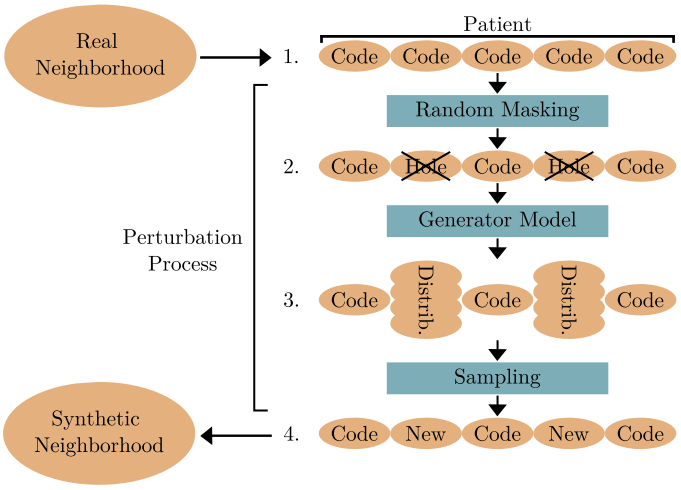

Figure 11: Pipeline of the generative process.

3. An output layer which is a simple linear function that shifts the dimension of the data from $n \times h$, which is the output of the transformer, to $n \times |\mathcal{C}|$.

4. A final SoftMax function, which converts the logits of the output layer into vector of probability over the codes.

The procedure for generating a synthetic patient from a real one p_r is shown in Figure 11 and is described as follows:

1. The codes of p_r are converted to their flattened form, and a vector of positional encodings q is built during the process.

2. Some codes chosen at random in the flattened patient are substituted with the `hole` code.

3. The noisy patient is fed to the generative model. For each code in input we have a distribution of probability over the possible codes that can be placed in that position.

4. New codes are sampled from the distributions corresponding to the `hole` codes. The sampled elements will substitute the `hole`s to obtain the final generated patient.

The objective of the generative model is to correctly denoise its input patients. Thus for the training we selected two distinct possible losses to use: a multi-label categorical cross-entropy over all the codes l_{all}, masked or not, and a similar multi-label categorical cross-entropy computed only on the

masked codes l_{masked}. Thus the final loss we used is a combination of the two through an hyperparameter λ_{loss}:

$$\text{Loss} = l_{\text{all}} + \lambda_{\text{loss}} l_{\text{masked}}.$$

4.3 Analysis of Attention

Given the wide spread of transformer architectures, we want to try to understand some of their components in the new light of our local explanations. We notice that to each code present in the patient to be explained we can associate a vector of attention weights for each self-attention head. In fact an attention scalar value associated to a given code is computed for each query. We can then aggregate all of these attention values with a permutation-invariant function. In our experiments the maximum, the minimum, and the arithmetic mean have been used.

With this process we can assign an attention score to each code that depends only on the predictive model, obtaining a vector of importance with an entry for each CCS code.

On the other hand, we can extract another kind of feature importance derived from the explanation tree, as presented in section 2.3.1.

We want to investigate whether the two concepts of importance agree with each other. To do so we compute a linear correlation coefficient between the two vectors. We observe that the vector with the attentions importances has to have zero entries on the codes not present in the patients to be explained, by construction. This is not the case for the feature importances from the tree, however we expect a good explainer to concentrate on the codes present in the data to be explained. Thus we expect the two vectors to agree assigning zero importance on a trivial subset of codes.

For this reason we also study the correlation between the attention and the tree-based feature importances when restricted to the codes present in the original patient.

We notice that while the attentions and the feature importances are similar measure of relevance for the prediction, they capture slightly different concepts. In fact the tree importances are computed with respect to the local neighborhood. On the other hand the attentions are computed locally (because they refer to a single patient passed as input of the predictive model) but they are run through the whole logic of the model.

5 Experiments

All the experiments have been executed thanks to the computational resources of National Recovery and Resilience Plan (Piano Nazionale di Ripresa e Resilienza, PNRR) – Project: "SoBigData.it – Strengthening the Italian RI for Social Mining and Big Data Analytics" – Prot. IR0000013 – Avviso n. 3264 del 28/12/2021.

5.1 Mimic IV

Mimic IV [13] is a dataset containing medical informations resulted from the collaboration of the Beth Israeli Deaconess Medical Center (BIDMC) and the Massachusetts Institute of Technology (MIT). Data is gathered as part of the routine activities at BIDMC, and processed at MIT.

The dataset contains informations about patients accessing the services at the emergency department or Intensive Care Units (ICUs) between 2008 and 2019. Patients that were below age 18 at their first admission time were excluded. People known to require extra protection were excluded as well. Between the raw data sources and the final published dataset, a step of deidentification of the patients has been performed, removing all personal informations and translating by a random time all the events regarding every single patient.

The dataset is divided into three separate modules: `hosp`, `icu`, and `note`.

The `hosp` module stores information regarding patient transfers, billed events, medication prescription, medication administration, laboratory values, microbiology measurements, and provider orders. This is the main source of data of this project.

The `icu` module contains all the information collected by the MetaVision clinical information system for the ICU units.

The `note` module contains textual data of discharges, which gathers an in-depth summary of the patients history during their stays at the hospital, and a section about radiology data.

Our main interest is in the `hosp` module, which contains among other things a table of admissions and a table of diagnoses. The admissions is a table that associates a patient id, an admission id, several temporal coordinates for the admissions such as the admission, registration and discharge times. There are also information about the patient like language, insurance, race and marital status. We will use only the patient and admission id, and the relative ordering given by the admission time. This table contains 431,231 rows, each corresponding to a unique admission.

The diagnoses table is much larger, containing 4,756,326 rows, each with a unique diagnose. A diagnose is composed by a patient id and admission id, which correspond to the ones given in the admission table, a couple icd-version and icd-code, and a numerical priority ranging 1-39 which ranks the importance of the codes within the same visit.

ICD, the International Classification of Diseases, which is a system to classify diagnostic statements maintained by the World Health Organization (WHO). There are several versions of these codes, and Mimic-iv uses ICD-9 and ICD-10. The version is flagged in a specific column of the diagnoses table. A code may indicate signs, symptoms, abnormal findings, complaints, social circumstances, and external causes of injury or disease. In Table 1 are reported some examples of ICD-10 codes. ICD-9 codes are very similar.

Code	Description
A00.0	Cholera due to Vibrio cholerae 01, biovar cholerae
E66.0	Obesity due to excess calories
I22.0	Subsequent myocardial infarction of anterior wall
F32.0	Mild depressive episode
O80.0	Spontaneous vertex delivery
S81.0	Open wound of knee
W58	Bitten or struck by crocodile or alligator
Z56.3	Stressful work schedule
Z72.0	Tobacco use (Excl. tobacco dependence)

Table 1: Some examples of ICD-10 diagnoses codes from different sections.

In the diagnoses table of the Mimic-iv dataset there are 2,766,877 ICD-9 codes (58% of the total) and 1,989,449 ICD-10 codes (42% of the total). Considering codes with different versions of ICD categorization as different, the dataset contains 25,829 distinct codes, which means that each code appears on average 184 times in the database. Of course the codes are not evenly distributed among those present. Distribution of the number of codes is shown in Figure 12.

The admission table contains 180,733 distinct patients. However there are 101,198 patients (56% of the total) who are present in a single admission. We will filter those out because to make a prediction that can be compared with a ground truth at least two visits are necessary. A percentage of 92% of patients has no more than five visits.

A preprocess phase is done by deleting all the patients who have a single visit, and then a conversion from ICD-10 to ICD-9 is performed, such that

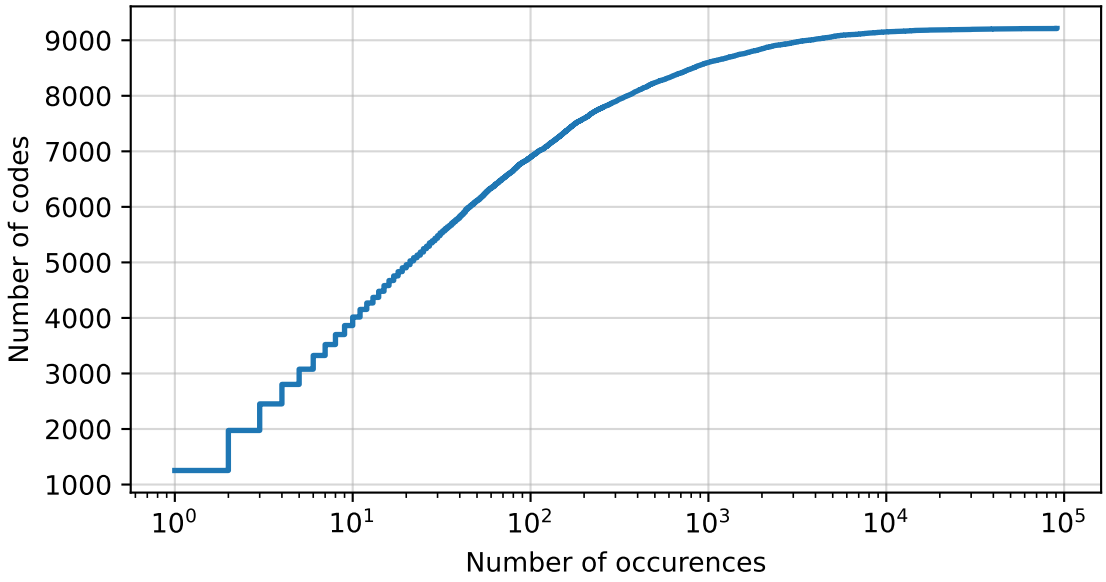

Figure 12: Distribution of the ICD codes. The x axis represent the number of occurrences of a given code in the dataset. The y axis show how many distinct codes have at most the corresponding number of occurrences. For example there are about 7,000 codes that occur less than 100 times.

all the models will work on a single ontology. The data that follows will be referred to the subset of patients with at least two visits in the dataset.

The conversion between the two versions of the codification is not straightforward, and some ICD-10 codes can be mapped to more than a ICD-9 code. Moreover, the conversion table we used does not contain all the ICD-10 codes present in Mimic-iv. These are the 0.8% of all the codes, and are assigned to a special "not found" class. When multiple conversions were available we chose a random one, as done in [22].

We also convert the ICD codes to another system called CCS (Clinical Classification Software) [1]. It is a coarser classification than ICD which has proven useful for analysis due to the lower number of codes. There are 281 distinct CCS codes within the dataset. Their distribution is shown in Figure 13.

5.2 Black-Box Model

Following the idea of other similar models [22], we use ICD codes as inputs and CCS codes as output. This allows us to have more accurate predictions

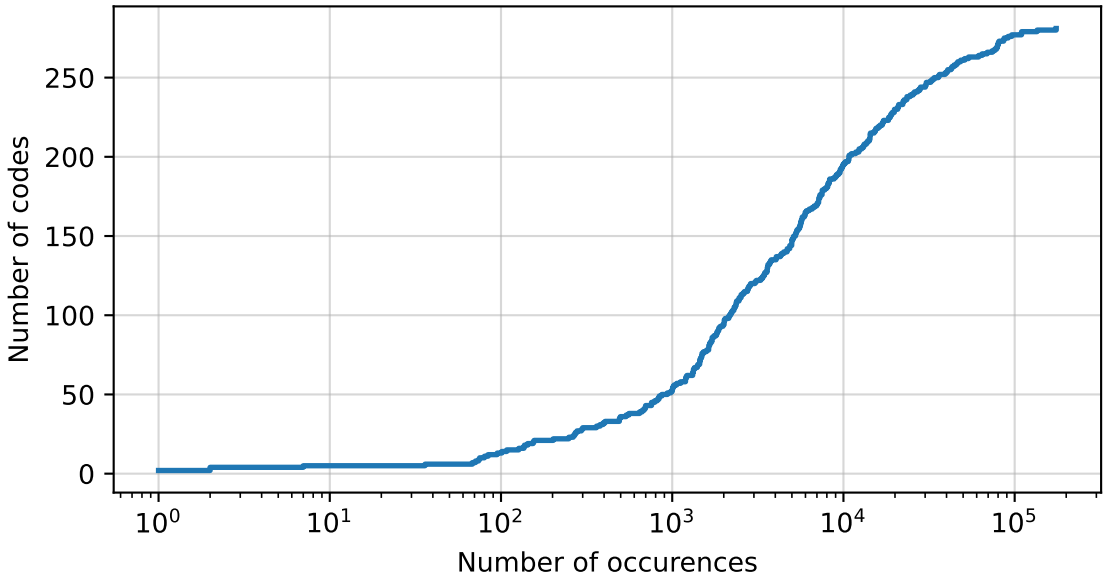

Figure 13: Distribution of the CCS codes. The x axis represent the number of occurrences of a given code in the dataset. The y axis show how many distinct codes have at most the corresponding number of occurrences. For example there are about 50 codes that occur less than 1,000 times.

than the case in which we used only CCS codes, while we keep a reasonable sized output dimension.

Our predictive model is trained with an Adam algorithm with Decoupled Weight Decay Regularization [16]. For each visit to be predicted the loss is computed as the code-by-code binary cross-entropy. This is then summed over the codes and averaged over all the visits of the minibatch.

We performed a hyperparameter selection with a complete grid search running the candidate models for a fixed amount of epochs, and choosing the best on the metrics obtained on a held-out validation set of the data. The following are the hyperparameter we used:

- *Hidden size.* This is the hidden size of the transformer, which is also equal to the size of the embedding vectors.
- *Number of layers.* It is the number of transformer layers present in the model.
- *Number of attention heads.* Number of self-attention heads in the transformer.

- *Attention head size*. This is the size of each key, query and value vector within each single self-attention head.
- *Learning rate*. This regulates the step size of the SGD algorithm.
- *Intermediate size of MLPs*, which is the size of the projection in the gated MLPs of the model
- *Parametrized pooling head*. It is a boolean value, which indicates whether to use the parametrized pooling head (when true), or the uniform one (when false).

We ran a complete hyperparameter search for choosing the best model, using the search space shown in Table 2. The search was performed by training each candidate model for 20 epochs, then we chose the model that showed the minimum loss on the evaluation set.

hyperparameter	Values
Hidden size	63, **128**, 256
Number of layers	1, **5**, 10
Number of attention heads	**4**, 8
Attention head size	15, **32**, 64
Learning rate	$\mathbf{9^{-3}}$, 10^{-4}
Intermediate size of MLPs	255, 512, **1024**
Parametrized pooling head	**True**, False

Table 2: hyperparameter search space for the black-box model. The selected parameters are shown in bold-face.

We trained the final model until the loss on the evaluation set stopped improving. The history of the loss over the evaluation set is shown in Figure 14. We also kept track of the recall metrics over the first 5, 10, 20 and 30 codes. The history of these metrics is shown in Figure 15. The results show that our model performs better than the version of doctorAI used in [20], but it still does not reach the performances of more complex models as SETOR [22], which also uses the ontology for the predictions. However one of the strengths of our model is its simplicity, which will allow us to perform the analysis described in the next sections. Table 3 compares the performances of these three different models.

We also tried training the model over a portion of the training set, while keeping the evaluation and tests set at their original sizes. Figure 16 shows the evaluation loss and recall metrics as a function of the size of the training set.

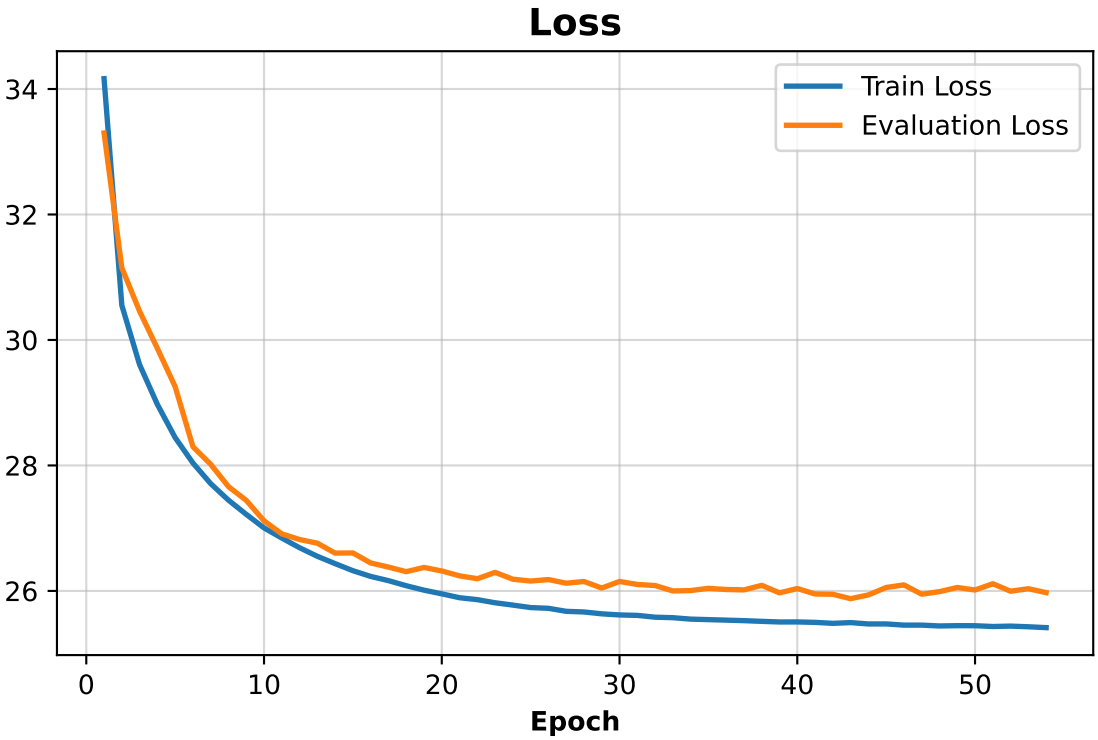

Figure 14: Loss history during the training of the predictor model.

5.3 Explanation Pipeline

The generative model must have ICD codes as outputs in order to be compatible with the black-box model. To keep the output layer at a reasonable dimension we set the target codes to be slightly different by the original patient to be reconstructed: we masked all the codes that are not between the k_{out} most frequent codes in the dataset with a special "`other`" code. The outputs sampled to be "`other`" are then left blank in the final generated patient.

We train the generative model in the same way as done for the predictor. We set the number of possible output codes k_{out} to 500. The codes to be replace with `hole` are chosen independently with probability p_{hole}. In our experiments we used a value of p_{hole} equal to 0.15.

We generated the synthetic neighborhoods from the real neighborhoods in five distinct ways. The first is the ontological perturbation described in [20]. Then we used the generative perturbations with two different methods, one guided by the generative model, and one with the new codes sampled

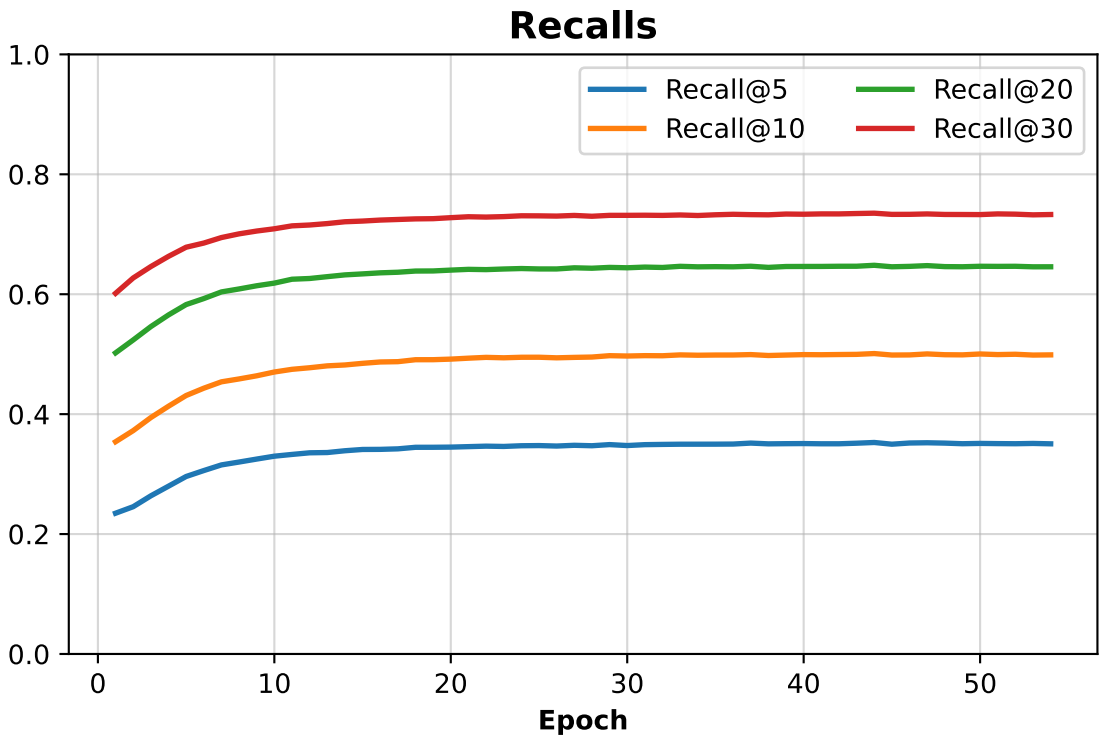

Figure 15: Recalls history during the training of the predictor model.

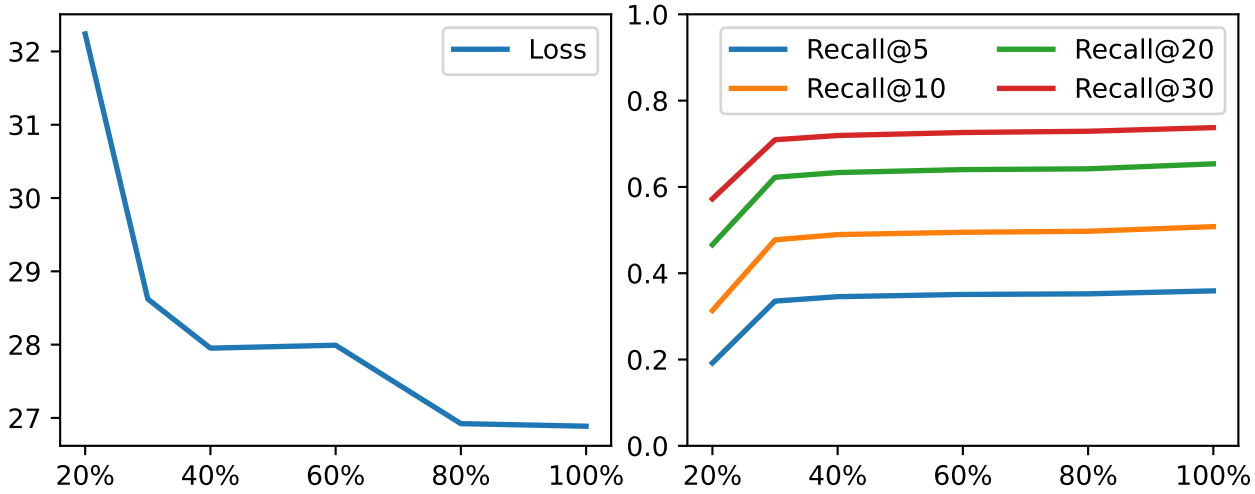

Figure 16: Metrics on the test set for the model trained on partial data. The percentage on the x-axis indicates the portion of the train set used.

Model	Recall@10	Recall@20	Recall@30
doctorAI	35.0%	52.1%	63.1%
Our model	50.8%	65.3%	73.7%
SETOR	70.6%	84.1%	88.2%

Table 3: Comparison of the performances of different models.

uniformly. Finally we used a combination of the two, running the generative perturbation on top of the perturbations generated by the ontological pipeline. Listing 1 shows an example of a patient with a prediction and the relative generated explanation in textual form. We first list the codes occurred at each visit and then we show the top 10 codes predicted by the black box. We then list the rules extracted by the decision tree classifier which consist in a code and threshold value, together with the value in the encoded patient instance to be explained. The codes in the patient that intervene in the decision rules are finally listed at the end of the explanation output. These are the codes that are important to understand the local logic of the black-box and should help a physician to understand the given prediction.

Listing 1: Example of patient with prediction and explanation

```
visit 1
  [78552]  Septic shock
  [2764]   Mixed acid-base balance disorder
  [V1551]  Personal history of traumatic fracture
  [7854]   Gangrene
  [6038]   Other specified types of hydrocele
  [60883]  Vascular disorders of male genital organs
  [79092]  Abnormal coagulation profile
  [0389]   Unspecified septicemia
  [2762]   Acidosis
  [99592]  Severe sepsis
  [2858]   Other specified anemias
  [07811]  Condyloma acuminatum
  [45829]  Other iatrogenic hypotension
  [72886]  Necrotizing fasciitis
  [5849]   Acute kidney failure, unspecified
visit 2
  [57451]  Calculus of bile duct without mention of cholecystitis, with obstruction
  [60000]  Hypertrophy (benign) of prostate without urinary obstruction and
             other lower urinary tract symptom (LUTS)
  [1970]   Secondary malignant neoplasm of lung
  [V5861]  Long-term (current) use of anticoagulants
  [03840]  Septicemia due to gram-negative organism, unspecified
  [V1006]  Personal history of malignant neoplasm of rectum,
             rectosigmoid junction, and anus
  [5849]   Acute kidney failure, unspecified
  [99592]  Severe sepsis
  [2859]   Anemia, unspecified
  [V1251]  Personal history of venous thrombosis and embolism
```

```
  [2449]   Unspecified acquired hypothyroidism
  [2762]   Acidosis
  [4010]   Malignant essential hypertension
visit 3
  [1970]   Secondary malignant neoplasm of lung
  [V5861]  Long-term (current) use of anticoagulants
  [4010]   Malignant essential hypertension
  [1541]   Malignant neoplasm of rectum
  [57451]  Calculus of bile duct without mention of cholecystitis, with obstruction
  [V443]   Colostomy status
  [2638]   Other protein-calorie malnutrition
  [V4572]  Acquired absence of intestine (large) (small)
  [V8523]  Body Mass Index 27.0-27.9, adult
  [53081]  Esophageal reflux
  [V1251]  Personal history of venous thrombosis and embolism
  [2859]   Anemia, unspecified
  [2449]   Unspecified acquired hypothyroidism
  [2768]   Hypopotassemia
  [60000]  Hypertrophy (benign) of prostate without urinary obstruction and
              other lower urinary tract symptom (LUTS)

ccs predicted
[118]   Phlebitis
[259]   Unclassified
[99]    Htn complicn
[155]   Other GI dx
[164]   BPH
[48]    Thyroid dsor
[15]    Rctm/anus ca
[42]    2ndary malig
[257]   Ot aftercare
[58]    Ot nutrit dx

decision rules
code 48   - threashold: 0.25, found 1.50,  (Thyroid dsor)
code 99   - threashold: 0.25, found 1.50,  (Htn complicn)
code 58   - threashold: 0.75, found 1.00,  (Ot nutrit dx)
code 118  - threashold: 0.25, found 1.50,  (Phlebitis)
code 52   - threashold: 0.50, found 1.00,  (Nutrit defic)

codes of the patient relevant for explanation
At visit 2, 3,  code V422   [Heart valve replaced by transplant]
At visit 2, 3,  code 9951   [Angioneurotic edema, not elsewhere classified]
At visit 2, 3,  code 2689   [Unspecified vitamin D deficiency]
At visit 3,     code 9951   [Angioneurotic edema, not elsewhere classified]
At visit 3,     code V1271  [Personal history of peptic ulcer disease]
At visit 3,     code 43300  [Occlusion and stenosis of basilar artery
                                without mention of cerebral infarction]
```

Across the various combinations we kept the size of the synthetic neighborhood fixed at 32,000 patients. In the case of a single type of perturbation (ontological or generative) we generated n_{single} artificial patient from each real one. In the case in which we used both types of perturbations, we generated n_{ont} synthetic patients from each real one, and then n_{gen} patients from each on of the previously generated. Table 4 shows the multiplication factors we used for each size of the real neighborhood. We notice that in some cases in the mixed generative method there are some unbalances between n_{ont} and

n_{gen}, where we always favored the generative part giving it more weight in the perturbation. We'll show however that despite this unbalance the results for the mixed generation methods are more similar to the ones obtained from the purely ontological perturbation.

Real neighborhood size	n_{single}	n_{ont}	n_{gen}
100	32	4	8
200	16	4	4
400	8	2	4
800	4	2	2

Table 4: Number of synthetic patients generated from each original real patient.

We run the experiments for the different settings computing the fidelity with the black-box. The results are shown in the first columns of Table 5. We notice that the fidelity doesn't change much across the real neighborhood size and various perturbation methods, except that it is slightly higher when the ontological perturbation is not used.

We proceed then to conduct the analysis described in Section 4.3. In fact, the other columns of Table 5 show the correlation between the attention and the feature importance extracted from the trees of the explanation. Table 6 shows the correlation when restricted to the codes present in the input.

The result show the "trivial correlation" on the importances associated to the all the codes. The correlation is always stronger when the aggregation function on the attention matrices is the Max function. When we restrict the analysis to only the input codes, the correlation with the Min and Average aggregation disappears, and the correlation with the Max aggregation is very weak.

More data on the correlation found at the various layer of the predictor model can be found in Appendix A.

Similar results can be found in [12], where attention fails to capture importances computed with gradient-based methods. Our results confirm that this happens also when attention are compared to local importances.

Neighs	Ont.	Gen.	Uni.	Fidelity	Corr/Max	Corr/Min	Corr/Avg
100	Yes	No	No	80.1%	0.61	-0.59	0.52
100	No	Yes	No	81.6%	0.49	-0.47	0.43
100	No	Yes	Yes	81.6%	0.50	-0.48	0.43
100	Yes	Yes	No	79.8%	0.58	-0.57	0.50
100	Yes	Yes	Yes	79.9%	0.58	-0.57	0.50
200	Yes	No	No	80.1%	0.60	-0.58	0.52
200	No	Yes	No	81.8%	0.52	-0.51	0.46
200	No	Yes	Yes	81.8%	0.53	-0.51	0.46
200	Yes	Yes	No	79.8%	0.59	-0.57	0.51
200	Yes	Yes	Yes	79.9%	0.59	-0.57	0.51
400	Yes	No	No	79.8%	0.58	-0.57	0.50
400	No	Yes	No	81.7%	0.54	-0.52	0.47
400	No	Yes	Yes	81.8%	0.54	-0.53	0.47
400	Yes	Yes	No	79.6%	0.57	-0.56	0.49
400	Yes	Yes	Yes	79.7%	0.57	-0.56	0.49
800	Yes	No	No	79.5%	0.56	-0.55	0.48
800	No	Yes	No	81.4%	0.54	-0.52	0.47
800	No	Yes	Yes	81.4%	0.54	-0.52	0.46
800	Yes	Yes	No	79.3%	0.56	-0.55	0.48
800	Yes	Yes	Yes	79.3%	0.55	-0.55	0.48

Table 5: Results of the explanation pipeline. *Neighs* is the size of the real neighborhood, *Ont.* shows whether or not the ontological perturbations are performed. *Gen.* shows whether or not the generative perturbations are performed. *Uni.* shows whether the generative perturbations were done with the transformer (value is No) or with uniform sampling (value is Yes). The fidelity reports the fidelity of the tree model using the black-box predictor as ground-truth. The three *Corr* fields show the correlation between the importances in the explanation and an aggregation of the attention in the transformer over all layers and all codes. The three aggregations are the maximum (*Max*), the minimum (*Min*), and the average (*Avg*).

Neighs	Ont.	Gen.	Uni.	Fidelity	Corr/Max	Corr/Min	Corr/Avg
100	Yes	No	No	80.0%	0.29	-0.08	0.03
100	No	Yes	No	81.6%	0.24	-0.07	0.03
100	No	Yes	Yes	81.8%	0.24	-0.06	0.04
100	Yes	Yes	No	79.9%	0.28	-0.08	0.03
100	Yes	Yes	Yes	79.7%	0.28	-0.09	0.04
200	Yes	No	No	80.1%	0.31	-0.10	0.03
200	No	Yes	No	81.8%	0.25	-0.06	0.03
200	No	Yes	Yes	81.7%	0.26	-0.06	0.04
200	Yes	Yes	No	79.9%	0.28	-0.08	0.02
200	Yes	Yes	Yes	79.9%	0.28	-0.08	0.01
400	Yes	No	No	79.8%	0.29	-0.09	0.02
400	No	Yes	No	81.7%	0.28	-0.08	0.04
400	No	Yes	Yes	81.6%	0.26	-0.07	0.03
400	Yes	Yes	No	79.5%	0.28	-0.11	0.01
400	Yes	Yes	Yes	79.6%	0.27	-0.10	0.01
800	Yes	No	No	79.6%	0.26	-0.11	0.01
800	No	Yes	No	81.3%	0.29	-0.07	0.04
800	No	Yes	Yes	81.3%	0.26	-0.07	0.03
800	Yes	Yes	No	79.3%	0.27	-0.10	0.00
800	Yes	Yes	Yes	79.4%	0.26	-0.12	-0.00

Table 6: This table shows the same kind of data of Table 5 where the correlations are computed only over the codes present in the original patient to explain.

6 Conclusions

Machine Learning tools have been proven to very useful and effective in the last years. When applied to the medical field good predictors can improve the decision making process of professional figures. However, to increase the usefulness and trustworthiness of these models, predictions should be accompanied by an explanation that describe the logic behind them. Explanation methods can be useful also to investigate biases, limits and internal functioning of black-box models.

We have built a predictor for the diagnoses table of the Mimic-iv dataset. We reached good performances while using models that are much simpler than those at the state of the art, allowing us to extract the attention matrices and easily attribute them directly to the codes in input. Our model is also an interesting example of how transformers can be used to process data that is more complex than its original use case, a stream of tokens.

We attached our new model to the ontological pipeline presented in [20], rewriting all the code to make a more efficient usage of the available hardware than the original. This allowed us to work smoothly with a dataset one order of a magnitude larger than the one used in the previous version.

We presented a new way to augment a real neighborhood with synthetic data points. The results of our experiments show that adding new codes with a method aware of the data distribution is no better than adding codes from a uniform distribution under the fidelity metric.

We leveraged the simplicity of our model to extract another kind of explanation from the attention weights in the transformer. We showed that those scores agree very weakly with the importances given by our post-hoc explanations. This expands a set of very similar results in [12] that indicate that the attention matrices are not good explainers and should not be used as an explanation mechanism.

A Analysis of Attention on Specific Layers

In this appendix we report more data similar to the one in Table 5 and Table 6. In these new tables the attention vectors are computed by aggregating only the values coming from a specific layer of the predictive model.

Ont. shows whether or not the ontological perturbations are performed. *Gen.* shows whether or not the generative perturbations are performed. *Uni.* shows whether the generative perturbations were done with the transformer (value is No) or with uniform sampling (value is Yes). The fidelity reports the fidelity of the tree model using the black-box predictor as ground-truth. The layer field indicates the layer from which the attention values are extracted or if they are extracted from all the layers. The three *Corr* fields show the correlation between the importances in the explanation and an aggregation of the attention in the transformer over all layers and all codes. The three aggregations are the maximum (*Max*), the minimum (*Min*), and the average (*Avg*).

Neighs	Ont.	Gen.	Uni.	Layer	Fidelity	Corr/Max	Corr/Min	Corr/Avg
100	Yes	No	No	80.1%	all	0.61	-0.59	0.52
100	Yes	No	No	80.1%	layer 0	0.60	-0.59	0.55
100	Yes	No	No	80.1%	layer 1	0.54	-0.59	0.48
100	Yes	No	No	80.1%	layer 2	0.56	-0.59	0.48
100	Yes	No	No	80.1%	layer 3	0.58	-0.59	0.51
100	Yes	No	No	80.1%	layer 4	0.54	-0.59	0.48
100	No	Yes	No	81.6%	all	0.49	-0.47	0.43
100	No	Yes	No	81.6%	layer 0	0.47	-0.47	0.44
100	No	Yes	No	81.6%	layer 1	0.44	-0.47	0.40
100	No	Yes	No	81.6%	layer 2	0.46	-0.47	0.40
100	No	Yes	No	81.6%	layer 3	0.46	-0.47	0.41
100	No	Yes	No	81.6%	layer 4	0.43	-0.47	0.39
100	No	Yes	Yes	81.6%	all	0.50	-0.48	0.43
100	No	Yes	Yes	81.6%	layer 0	0.48	-0.48	0.44
100	No	Yes	Yes	81.6%	layer 1	0.45	-0.48	0.40
100	No	Yes	Yes	81.6%	layer 2	0.46	-0.48	0.41
100	No	Yes	Yes	81.6%	layer 3	0.47	-0.48	0.42
100	No	Yes	Yes	81.6%	layer 4	0.44	-0.48	0.40
100	Yes	Yes	No	79.8%	all	0.58	-0.57	0.50
100	Yes	Yes	No	79.8%	layer 0	0.58	-0.57	0.52
100	Yes	Yes	No	79.8%	layer 1	0.52	-0.57	0.47
100	Yes	Yes	No	79.8%	layer 2	0.53	-0.57	0.47
100	Yes	Yes	No	79.8%	layer 3	0.55	-0.57	0.48
100	Yes	Yes	No	79.8%	layer 4	0.51	-0.57	0.46
100	Yes	Yes	Yes	79.9%	all	0.58	-0.57	0.50
100	Yes	Yes	Yes	79.9%	layer 0	0.58	-0.57	0.53
100	Yes	Yes	Yes	79.9%	layer 1	0.52	-0.57	0.47
100	Yes	Yes	Yes	79.9%	layer 2	0.53	-0.57	0.47
100	Yes	Yes	Yes	79.9%	layer 3	0.55	-0.57	0.49
100	Yes	Yes	Yes	79.9%	layer 4	0.52	-0.57	0.47

Neighs	Ont.	Gen.	Uni.	Layer	Fidelity	Corr/Max	Corr/Min	Corr/Avg
200	Yes	No	No	80.1%	all	0.60	-0.58	0.52
200	Yes	No	No	80.1%	layer 0	0.60	-0.58	0.54
200	Yes	No	No	80.1%	layer 1	0.53	-0.58	0.47
200	Yes	No	No	80.1%	layer 2	0.55	-0.58	0.48
200	Yes	No	No	80.1%	layer 3	0.57	-0.58	0.50
200	Yes	No	No	80.1%	layer 4	0.54	-0.58	0.48
200	No	Yes	No	81.8%	all	0.52	-0.51	0.46
200	No	Yes	No	81.8%	layer 0	0.51	-0.51	0.47
200	No	Yes	No	81.8%	layer 1	0.47	-0.51	0.42
200	No	Yes	No	81.8%	layer 2	0.48	-0.51	0.42
200	No	Yes	No	81.8%	layer 3	0.50	-0.51	0.44
200	No	Yes	No	81.8%	layer 4	0.46	-0.51	0.42
200	No	Yes	Yes	81.8%	all	0.53	-0.51	0.46
200	No	Yes	Yes	81.8%	layer 0	0.51	-0.51	0.47
200	No	Yes	Yes	81.8%	layer 1	0.47	-0.51	0.43
200	No	Yes	Yes	81.8%	layer 2	0.48	-0.51	0.43
200	No	Yes	Yes	81.8%	layer 3	0.50	-0.51	0.45
200	No	Yes	Yes	81.8%	layer 4	0.47	-0.51	0.42
200	Yes	Yes	No	79.8%	all	0.59	-0.57	0.51
200	Yes	Yes	No	79.8%	layer 0	0.58	-0.57	0.53
200	Yes	Yes	No	79.8%	layer 1	0.52	-0.57	0.47
200	Yes	Yes	No	79.8%	layer 2	0.54	-0.57	0.47
200	Yes	Yes	No	79.8%	layer 3	0.56	-0.57	0.49
200	Yes	Yes	No	79.8%	layer 4	0.52	-0.57	0.47
200	Yes	Yes	Yes	79.9%	all	0.59	-0.57	0.51
200	Yes	Yes	Yes	79.9%	layer 0	0.58	-0.57	0.53
200	Yes	Yes	Yes	79.9%	layer 1	0.52	-0.57	0.47
200	Yes	Yes	Yes	79.9%	layer 2	0.54	-0.57	0.47
200	Yes	Yes	Yes	79.9%	layer 3	0.56	-0.57	0.49
200	Yes	Yes	Yes	79.9%	layer 4	0.52	-0.57	0.47

Neighs	Ont.	Gen.	Uni.	Layer	Fidelity	Corr/Max	Corr/Min	Corr/Avg
400	Yes	No	No	79.8%	all	0.58	-0.57	0.50
400	Yes	No	No	79.8%	layer 0	0.57	-0.57	0.52
400	Yes	No	No	79.8%	layer 1	0.51	-0.57	0.46
400	Yes	No	No	79.8%	layer 2	0.53	-0.57	0.46
400	Yes	No	No	79.8%	layer 3	0.55	-0.57	0.48
400	Yes	No	No	79.8%	layer 4	0.51	-0.57	0.46
400	No	Yes	No	81.7%	all	0.54	-0.52	0.47
400	No	Yes	No	81.7%	layer 0	0.53	-0.52	0.48
400	No	Yes	No	81.7%	layer 1	0.48	-0.52	0.43
400	No	Yes	No	81.7%	layer 2	0.49	-0.52	0.43
400	No	Yes	No	81.7%	layer 3	0.51	-0.52	0.45
400	No	Yes	No	81.7%	layer 4	0.47	-0.52	0.43
400	No	Yes	Yes	81.8%	all	0.54	-0.53	0.47
400	No	Yes	Yes	81.8%	layer 0	0.53	-0.53	0.49
400	No	Yes	Yes	81.8%	layer 1	0.48	-0.53	0.43
400	No	Yes	Yes	81.8%	layer 2	0.49	-0.53	0.44
400	No	Yes	Yes	81.8%	layer 3	0.52	-0.53	0.46
400	No	Yes	Yes	81.8%	layer 4	0.48	-0.53	0.44
400	Yes	Yes	No	79.6%	all	0.57	-0.56	0.49
400	Yes	Yes	No	79.6%	layer 0	0.56	-0.56	0.51
400	Yes	Yes	No	79.6%	layer 1	0.50	-0.56	0.45
400	Yes	Yes	No	79.6%	layer 2	0.51	-0.56	0.45
400	Yes	Yes	No	79.6%	layer 3	0.54	-0.56	0.48
400	Yes	Yes	No	79.6%	layer 4	0.50	-0.56	0.45
400	Yes	Yes	Yes	79.7%	all	0.57	-0.56	0.49
400	Yes	Yes	Yes	79.7%	layer 0	0.57	-0.56	0.52
400	Yes	Yes	Yes	79.7%	layer 1	0.50	-0.56	0.45
400	Yes	Yes	Yes	79.7%	layer 2	0.52	-0.56	0.45
400	Yes	Yes	Yes	79.7%	layer 3	0.54	-0.56	0.48
400	Yes	Yes	Yes	79.7%	layer 4	0.50	-0.56	0.46

Neighs	Ont.	Gen.	Uni.	Layer	Fidelity	Corr/Max	Corr/Min	Corr/Avg
800	Yes	No	No	79.5%	all	0.56	-0.55	0.48
800	Yes	No	No	79.5%	layer 0	0.56	-0.55	0.51
800	Yes	No	No	79.5%	layer 1	0.49	-0.55	0.44
800	Yes	No	No	79.5%	layer 2	0.51	-0.55	0.44
800	Yes	No	No	79.5%	layer 3	0.53	-0.55	0.47
800	Yes	No	No	79.5%	layer 4	0.49	-0.55	0.45
800	No	Yes	No	81.4%	all	0.54	-0.52	0.47
800	No	Yes	No	81.4%	layer 0	0.53	-0.52	0.48
800	No	Yes	No	81.4%	layer 1	0.47	-0.52	0.43
800	No	Yes	No	81.4%	layer 2	0.49	-0.52	0.43
800	No	Yes	No	81.4%	layer 3	0.51	-0.52	0.45
800	No	Yes	No	81.4%	layer 4	0.47	-0.52	0.43
800	No	Yes	Yes	81.4%	all	0.54	-0.52	0.46
800	No	Yes	Yes	81.4%	layer 0	0.53	-0.52	0.48
800	No	Yes	Yes	81.4%	layer 1	0.47	-0.52	0.43
800	No	Yes	Yes	81.4%	layer 2	0.48	-0.52	0.43
800	No	Yes	Yes	81.4%	layer 3	0.51	-0.52	0.45
800	No	Yes	Yes	81.4%	layer 4	0.47	-0.52	0.43
800	Yes	Yes	No	79.3%	all	0.56	-0.55	0.48
800	Yes	Yes	No	79.3%	layer 0	0.55	-0.55	0.51
800	Yes	Yes	No	79.3%	layer 1	0.49	-0.55	0.44
800	Yes	Yes	No	79.3%	layer 2	0.51	-0.55	0.44
800	Yes	Yes	No	79.3%	layer 3	0.53	-0.55	0.46
800	Yes	Yes	No	79.3%	layer 4	0.49	-0.55	0.45
800	Yes	Yes	Yes	79.3%	all	0.55	-0.55	0.48
800	Yes	Yes	Yes	79.3%	layer 0	0.55	-0.55	0.50
800	Yes	Yes	Yes	79.3%	layer 1	0.48	-0.55	0.43
800	Yes	Yes	Yes	79.3%	layer 2	0.50	-0.55	0.44
800	Yes	Yes	Yes	79.3%	layer 3	0.53	-0.55	0.46
800	Yes	Yes	Yes	79.3%	layer 4	0.49	-0.55	0.45

The following tables are relative to the attention computed only over the input codes, as described in section 4.3.

Neighs	Ont.	Gen.	Uni.	Layer	Fidelity	Corr/Max	Corr/Min	Corr/Avg
100	Yes	No	No	80.0%	all	0.29	-0.08	0.03
100	Yes	No	No	80.0%	layer 0	0.33	-0.05	0.07
100	Yes	No	No	80.0%	layer 1	0.21	-0.07	0.04
100	Yes	No	No	80.0%	layer 2	0.23	-0.13	0.02
100	Yes	No	No	80.0%	layer 3	0.26	-0.15	0.03
100	Yes	No	No	80.0%	layer 4	0.20	-0.16	0.01
100	No	Yes	No	81.6%	all	0.24	-0.07	0.03
100	No	Yes	No	81.6%	layer 0	0.23	-0.04	0.04
100	No	Yes	No	81.6%	layer 1	0.18	-0.04	0.05
100	No	Yes	No	81.6%	layer 2	0.19	-0.09	0.03
100	No	Yes	No	81.6%	layer 3	0.20	-0.09	0.03
100	No	Yes	No	81.6%	layer 4	0.16	-0.11	0.02
100	No	Yes	Yes	81.8%	all	0.24	-0.06	0.04
100	No	Yes	Yes	81.8%	layer 0	0.22	-0.03	0.04
100	No	Yes	Yes	81.8%	layer 1	0.20	-0.03	0.06
100	No	Yes	Yes	81.8%	layer 2	0.19	-0.07	0.04
100	No	Yes	Yes	81.8%	layer 3	0.22	-0.10	0.04
100	No	Yes	Yes	81.8%	layer 4	0.16	-0.12	0.01
100	Yes	Yes	No	79.9%	all	0.28	-0.08	0.03
100	Yes	Yes	No	79.9%	layer 0	0.31	-0.05	0.07
100	Yes	Yes	No	79.9%	layer 1	0.19	-0.09	0.02
100	Yes	Yes	No	79.9%	layer 2	0.21	-0.13	0.01
100	Yes	Yes	No	79.9%	layer 3	0.26	-0.15	0.03
100	Yes	Yes	No	79.9%	layer 4	0.20	-0.14	0.02
100	Yes	Yes	Yes	79.7%	all	0.28	-0.09	0.04
100	Yes	Yes	Yes	79.7%	layer 0	0.33	-0.05	0.08
100	Yes	Yes	Yes	79.7%	layer 1	0.21	-0.07	0.05
100	Yes	Yes	Yes	79.7%	layer 2	0.22	-0.13	0.02
100	Yes	Yes	Yes	79.7%	layer 3	0.26	-0.15	0.04
100	Yes	Yes	Yes	79.7%	layer 4	0.20	-0.14	0.03

Neighs	Ont.	Gen.	Uni.	Layer	Fidelity	Corr/Max	Corr/Min	Corr/Avg
200	Yes	No	No	80.1%	all	0.31	-0.10	0.03
200	Yes	No	No	80.1%	layer 0	0.34	-0.05	0.08
200	Yes	No	No	80.1%	layer 1	0.22	-0.09	0.04
200	Yes	No	No	80.1%	layer 2	0.24	-0.14	0.02
200	Yes	No	No	80.1%	layer 3	0.27	-0.17	0.02
200	Yes	No	No	80.1%	layer 4	0.21	-0.16	0.01
200	No	Yes	No	81.8%	all	0.25	-0.06	0.03
200	No	Yes	No	81.8%	layer 0	0.25	-0.03	0.05
200	No	Yes	No	81.8%	layer 1	0.19	-0.06	0.05
200	No	Yes	No	81.8%	layer 2	0.20	-0.09	0.03
200	No	Yes	No	81.8%	layer 3	0.23	-0.12	0.02
200	No	Yes	No	81.8%	layer 4	0.17	-0.12	0.02
200	No	Yes	Yes	81.7%	all	0.26	-0.06	0.04
200	No	Yes	Yes	81.7%	layer 0	0.26	-0.04	0.06
200	No	Yes	Yes	81.7%	layer 1	0.18	-0.04	0.05
200	No	Yes	Yes	81.7%	layer 2	0.20	-0.09	0.03
200	No	Yes	Yes	81.7%	layer 3	0.24	-0.13	0.04
200	No	Yes	Yes	81.7%	layer 4	0.19	-0.12	0.02
200	Yes	Yes	No	79.9%	all	0.28	-0.08	0.02
200	Yes	Yes	No	79.9%	layer 0	0.33	-0.05	0.08
200	Yes	Yes	No	79.9%	layer 1	0.20	-0.08	0.03
200	Yes	Yes	No	79.9%	layer 2	0.23	-0.13	0.01
200	Yes	Yes	No	79.9%	layer 3	0.27	-0.18	0.02
200	Yes	Yes	No	79.9%	layer 4	0.21	-0.13	0.01
200	Yes	Yes	Yes	79.9%	all	0.28	-0.08	0.01
200	Yes	Yes	Yes	79.9%	layer 0	0.32	-0.05	0.06
200	Yes	Yes	Yes	79.9%	layer 1	0.18	-0.09	0.01
200	Yes	Yes	Yes	79.9%	layer 2	0.22	-0.15	-0.00
200	Yes	Yes	Yes	79.9%	layer 3	0.27	-0.17	0.02
200	Yes	Yes	Yes	79.9%	layer 4	0.20	-0.15	0.01

Neighs	Ont.	Gen.	Uni.	Layer	Fidelity	Corr/Max	Corr/Min	Corr/Avg
400	Yes	No	No	79.8%	all	0.29	-0.09	0.02
400	Yes	No	No	79.8%	layer 0	0.33	-0.04	0.08
400	Yes	No	No	79.8%	layer 1	0.19	-0.09	0.02
400	Yes	No	No	79.8%	layer 2	0.22	-0.15	0.01
400	Yes	No	No	79.8%	layer 3	0.25	-0.19	0.01
400	Yes	No	No	79.8%	layer 4	0.21	-0.13	0.02
400	No	Yes	No	81.7%	all	0.28	-0.08	0.04
400	No	Yes	No	81.7%	layer 0	0.28	-0.03	0.07
400	No	Yes	No	81.7%	layer 1	0.19	-0.07	0.04
400	No	Yes	No	81.7%	layer 2	0.22	-0.11	0.02
400	No	Yes	No	81.7%	layer 3	0.26	-0.12	0.04
400	No	Yes	No	81.7%	layer 4	0.20	-0.11	0.03
400	No	Yes	Yes	81.6%	all	0.26	-0.07	0.03
400	No	Yes	Yes	81.6%	layer 0	0.27	-0.04	0.06
400	No	Yes	Yes	81.6%	layer 1	0.19	-0.06	0.04
400	No	Yes	Yes	81.6%	layer 2	0.20	-0.10	0.01
400	No	Yes	Yes	81.6%	layer 3	0.24	-0.12	0.03
400	No	Yes	Yes	81.6%	layer 4	0.18	-0.12	0.01
400	Yes	Yes	No	79.5%	all	0.28	-0.11	0.01
400	Yes	Yes	No	79.5%	layer 0	0.32	-0.03	0.08
400	Yes	Yes	No	79.5%	layer 1	0.18	-0.10	0.01
400	Yes	Yes	No	79.5%	layer 2	0.22	-0.15	0.00
400	Yes	Yes	No	79.5%	layer 3	0.26	-0.20	0.01
400	Yes	Yes	No	79.5%	layer 4	0.19	-0.14	0.00
400	Yes	Yes	Yes	79.6%	all	0.27	-0.10	0.01
400	Yes	Yes	Yes	79.6%	layer 0	0.32	-0.03	0.07
400	Yes	Yes	Yes	79.6%	layer 1	0.18	-0.10	0.00
400	Yes	Yes	Yes	79.6%	layer 2	0.21	-0.15	-0.01
400	Yes	Yes	Yes	79.6%	layer 3	0.25	-0.17	0.01
400	Yes	Yes	Yes	79.6%	layer 4	0.20	-0.15	0.01

Neighs	Ont.	Gen.	Uni.	Layer	Fidelity	Corr/Max	Corr/Min	Corr/Avg
800	Yes	No	No	79.6%	all	0.26	-0.11	0.01
800	Yes	No	No	79.6%	layer 0	0.33	-0.03	0.08
800	Yes	No	No	79.6%	layer 1	0.16	-0.10	-0.01
800	Yes	No	No	79.6%	layer 2	0.20	-0.16	-0.01
800	Yes	No	No	79.6%	layer 3	0.25	-0.15	0.01
800	Yes	No	No	79.6%	layer 4	0.19	-0.15	0.00
800	No	Yes	No	81.3%	all	0.29	-0.07	0.04
800	No	Yes	No	81.3%	layer 0	0.30	-0.03	0.08
800	No	Yes	No	81.3%	layer 1	0.20	-0.08	0.03
800	No	Yes	No	81.3%	layer 2	0.22	-0.11	0.02
800	No	Yes	No	81.3%	layer 3	0.27	-0.13	0.05
800	No	Yes	No	81.3%	layer 4	0.19	-0.11	0.02
800	No	Yes	Yes	81.3%	all	0.26	-0.07	0.03
800	No	Yes	Yes	81.3%	layer 0	0.30	-0.02	0.07
800	No	Yes	Yes	81.3%	layer 1	0.18	-0.06	0.03
800	No	Yes	Yes	81.3%	layer 2	0.20	-0.12	0.01
800	No	Yes	Yes	81.3%	layer 3	0.26	-0.14	0.04
800	No	Yes	Yes	81.3%	layer 4	0.17	-0.12	0.01
800	Yes	Yes	No	79.3%	all	0.27	-0.10	0.00
800	Yes	Yes	No	79.3%	layer 0	0.32	-0.03	0.08
800	Yes	Yes	No	79.3%	layer 1	0.17	-0.10	-0.01
800	Yes	Yes	No	79.3%	layer 2	0.19	-0.16	-0.02
800	Yes	Yes	No	79.3%	layer 3	0.25	-0.20	0.00
800	Yes	Yes	No	79.3%	layer 4	0.18	-0.14	-0.00
800	Yes	Yes	Yes	79.4%	all	0.26	-0.12	-0.00
800	Yes	Yes	Yes	79.4%	layer 0	0.33	-0.04	0.08
800	Yes	Yes	Yes	79.4%	layer 1	0.16	-0.11	-0.01
800	Yes	Yes	Yes	79.4%	layer 2	0.20	-0.18	-0.03
800	Yes	Yes	Yes	79.4%	layer 3	0.25	-0.20	-0.00
800	Yes	Yes	Yes	79.4%	layer 4	0.17	-0.16	-0.01

B Code

In this section we report the code we used to perform all the experiments in this thesis. All the code is also available at `https://github.com/Teiolass/tesi-magistrale-code`. All the code has be run using Python 3.11. For performance reasons, to generate the explanations we leverage on a low level library compiled with Zig 0.11. Precise instructions on how to run the code to replicate the experiments can be found at the page of the repo.

The following listing contains the code for preprocessing the data.

```
import polars as pl
from polars import col

import os

#### PARAMETERS

mimic_prefix  = 'data/mimic-iv/2.2/hosp/'
ccs_prefix    = 'data/ccs'
output_prefix = 'data/processed'

# these are the mimic-iv tables
diagnoses_file  = 'diagnoses_icd.csv.gz'
admissions_file = 'admissions.csv.gz'
# this is to convert icd10 to icd8
icd_conv_file   = 'icd10cmtoicd9gem.csv'
# this is for converting icd9 to ccs
ccs_single_file = 'ccs_single_dx_tool_2015.csv'
# this contains the ontology over the icds
ontology_file   = 'data/ICD9CM.csv'

# these are the output files
output_diagnoses  = 'diagnoses.parquet'
output_ccs        = 'ccs.parquet'
output_icd        = 'icd.parquet'
output_ontology   = 'ontology.parquet'
output_generation = 'generation.parquet'

# minimum number of icd occurences to not be assigned a 0 code
min_icd_occurences = 20
# this is for the generator model
num_output_codes = 500

train_fraction = 0.7
eval_fraction  = 0.15

#### END PARAMETERS

ontology_prefixes = ['http://purl.bioontology.org/ontology/ICD9CM/', 'http
    ↪ ://purl.bioontology.org/ontology/STY/']
ontology_root_name = 'root'

# load all the data
diagnoses_path = os.path.join(mimic_prefix, diagnoses_file)
diagnoses      = pl.read_csv(diagnoses_path, dtypes={'subject_id':pl.UInt64,
    ↪  'icd_version':pl.UInt8})
admission_path = os.path.join(mimic_prefix, admissions_file)
admissions     = pl.read_csv(admission_path)
```

```
icd_conv_path  = os.path.join(ccs_prefix, icd_conv_file)
icd_conv       = pl.read_csv(icd_conv_path, dtypes={'icd10cm':pl.Utf8, '
    ↪ icd9cm':pl.Utf8})
ccs_conv_path  = os.path.join(ccs_prefix, ccs_single_file)
ccs_conv       = pl.read_csv(ccs_conv_path, quote_char="'")

diagnoses = diagnoses[['subject_id', 'hadm_id', 'icd_code', 'icd_version']]
diagnoses = diagnoses.with_columns (
    visit_count = col('hadm_id').unique().count().over('subject_id')
).filter(col('visit_count') > 1)

icd_conv = icd_conv.lazy().select (
    col('icd10cm'),
    col('icd9cm').first().over('icd10cm')
).unique().collect()

# convert icd10 to icd9
diagnoses_icd10 = (
    diagnoses
        .filter(col('icd_version') == 10)
        .join(icd_conv, left_on='icd_code', right_on='icd10cm', how='left')
        .select(col('subject_id'), col('hadm_id'), col('icd9cm').alias('
            ↪ icd_code'))
)
diagnoses_icd9 = diagnoses.filter(col('icd_version') == 9)[diagnoses_icd10.
    ↪ columns]
diagnoses = pl.concat([diagnoses_icd9, diagnoses_icd10], how='vertical')

# convert icd9 to ccs
ccs_conv = ccs_conv.select (
    icd9 = col('ICD-9-CM␣CODE').str.strip_chars(),
    ccs  = col('CCS␣CATEGORY' ).str.strip_chars().cast(pl.UInt16),
    description = col('CCS␣CATEGORY␣DESCRIPTION')
)
diagnoses = (
    diagnoses
    .join(
        ccs_conv[['icd9', 'ccs']],
        left_on  = 'icd_code',
        right_on = 'icd9',
        how = 'left'
    )
    .with_columns (
        ccs      = col('ccs'     ).fill_null(pl.lit(0)),
        icd_code = col('icd_code').fill_null(pl.lit('NoDx'))
    )
).unique()

# convert ccs codes to ids
ccs_codes = diagnoses[['ccs']].unique().sort('ccs')
we_have_zero_code = 0 in ccs_codes['ccs'].head(1)
if not we_have_zero_code:
    print('WARNING:␣we␣do␣not␣have␣a␣zero␣code!')
ccs_codes = ccs_codes.join(ccs_conv.drop('icd9').unique('ccs'), on='ccs',
    ↪ how='left')
starting_index = 0 if we_have_zero_code else 1
indexes = pl.DataFrame({'ccs_id': range(starting_index, starting_index+
    ↪ ccs_codes.shape[0])}, schema={'ccs_id':pl.UInt32})
ccs_codes = pl.concat([ccs_codes, indexes], how='horizontal')
diagnoses = diagnoses.join(ccs_codes[['ccs', 'ccs_id']], on='ccs', how='left
    ↪ ').drop('ccs')
```

```
# convert icd9 codes to id
icd9_codes = diagnoses['icd_code'].value_counts()
num_codes_before = icd9_codes.shape[0]
icd9_codes = icd9_codes.filter(col('counts') >= min_icd_occurences).drop('
    ↪ counts')
print(
    f'There␣were␣{num_codes_before}␣different␣icd␣codes␣in␣the␣dataset.␣'
    f'We␣are␣keeping␣only␣those␣with␣at␣least␣{min_icd_occurences}.␣'
    f'There␣are␣{icd9_codes.shape[0]}␣icd␣codes␣remaining'
)
indexes = pl.DataFrame({'icd9_id': range(1, icd9_codes.shape[0]+1)}, schema
    ↪ ={'icd9_id':pl.UInt32})
icd9_codes = pl.concat([icd9_codes, indexes], how='horizontal')
icd9_codes = icd9_codes.with_columns(icd9_id=pl.when(pl.col('icd_code') == '
    ↪ NoDx').then(pl.lit(0)).otherwise(pl.col('icd9_id')))
diagnoses = (
    diagnoses
    .join (
        icd9_codes,
        on = 'icd_code',
        how = 'left',
    )
    .with_columns(icd9_id=col('icd9_id').fill_null(pl.lit(0)))
    .drop('icd_code')
)

# add time data
admissions = admissions.select (
    col('hadm_id'),
    col('admittime').str.to_datetime('%Y-%m-%d␣%H:%M:%S')
)
diagnoses = diagnoses.join(admissions, on='hadm_id', how='left').drop('
    ↪ hadm_id')


# this is for the generative model
top_codes = diagnoses['icd9_id'].value_counts().sort('counts', descending=
    ↪ True).head(num_output_codes)
top_codes = top_codes.filter(pl.col('icd9_id') != 0)
top_codes = top_codes.select(
    pl.col('icd9_id'),
    out_id = pl.arange(1, top_codes.shape[0]+1),
)
diagnoses = diagnoses.join(top_codes, on='icd9_id', how='left')
diagnoses = diagnoses.with_columns(pl.col('out_id').fill_null(0))
generation_conversion = top_codes.join(icd9_codes, on='icd9_id')
generation_conversion = generation_conversion.join(ccs_conv, left_on='
    ↪ icd_code', right_on='icd9', how='left')
generation_conversion = generation_conversion.join(ccs_codes, on='ccs', how=
    ↪ 'left')
generation_conversion = generation_conversion[['icd9_id', 'out_id', 'ccs_id'
    ↪ ]]


# prepare for use
diagnoses_a = (
    diagnoses
    .with_columns(
        position = col('admittime').rank('dense').over('subject_id') - 1,
    )
    .group_by('subject_id')
    .agg(
```

```
        col(['ccs_id', 'icd9_id', 'position', 'out_id']).sort_by('admittime'
            ↪ ),
    )
)
diagnoses_b = (
    diagnoses
    .group_by(['subject_id', 'admittime'])
    .agg(
        pl.count(),
     )
    .group_by('subject_id')
    .agg(col('count').sort_by('admittime'))
)
diagnoses = diagnoses_a.join(diagnoses_b, on='subject_id', how='inner')

# Split in train/eval/test

split_train = int(len(diagnoses) * train_fraction)
split_eval  = int(split_train + len(diagnoses) * eval_fraction)

diagnoses_cols = diagnoses.columns
diagnoses = (
    diagnoses.lazy()
    .with_row_count('ix')
    .with_columns(col('ix').shuffle())
    .select (
        diagnoses_cols,
        role = (
            pl
            .when(col('ix') < split_train)
            .then(pl.lit('train', dtype=pl.Categorical))
            .when(col('ix') < split_eval)
            .then(pl.lit('eval', dtype=pl.Categorical))
            .otherwise(pl.lit('test', dtype=pl.Categorical))
        )
    )
    .collect()
)

# Build Ontology

ontology   = pl.read_csv(ontology_file)

def remove_prefixes(exp: pl.Expr, prefixes: list[str]) -> pl.Expr:
    for p in prefixes:
        exp = exp.str.strip_prefix(p)
    return exp
ontology = ontology.lazy().select(
    label    = pl.col('Preferred␣Label'),
    icd_code = remove_prefixes(pl.col('Class␣ID'), ontology_prefixes),
    parent   = remove_prefixes(pl.col('Parents' ), ontology_prefixes),
).collect()

diagnoses_type = ontology.filter(pl.col('parent').str.starts_with('http') &
    ↪ (pl.col('label') != 'PROCEDURES'))['icd_code']

num_rows = 0
while num_rows != len(diagnoses_type):
    num_rows = len(diagnoses_type)
    t = pl.DataFrame({'parent': diagnoses_type})
    t = t.join(ontology[['parent', 'icd_code']], how='left', on='parent')
    diagnoses_type = pl.concat([t['icd_code'], diagnoses_type]).unique()
```

```
t = pl.DataFrame({'icd_code': diagnoses_type})
ontology = t.join(ontology, how='left', on='icd_code')

ontology = ontology.with_columns(
    icd_code = pl.col('icd_code').str.replace('.', '', literal=True),
    parent   = pl.when (
        pl.col('parent').str.starts_with('http')
    )
    .then(pl.lit(ontology_root_name))
    .otherwise(pl.col('parent').str.replace('.', '', literal=True).fill_null
        ↪ (pl.lit(ontology_root_name))),
)

ontology = ontology.join(icd9_codes, on='icd_code', how='outer').filter(~pl.
    ↪ col('icd_code').is_null())

# this assigns root as parent of NoDx
ontology = ontology.with_columns(pl.col('parent').fill_null(pl.lit(
    ↪ ontology_root_name)))

uncoded = ontology.filter(pl.col('icd9_id').is_null())
first_new_id = icd9_codes['icd9_id'].max() + 1
indexes = pl.DataFrame({'icd9_id': range(first_new_id, first_new_id+len(
    ↪ uncoded))}, schema={'icd9_id':pl.UInt32})
uncoded = pl.concat([uncoded.drop('icd9_id'), indexes], how='horizontal')

root_id = first_new_id + len(uncoded)
# schema is {'icd_code': Utf8, 'label': Utf8, 'parent': Utf8, 'icd9_id':
    ↪ UInt32}
root_row = pl.DataFrame(
    {'icd_code':ontology_root_name, 'label':'Root␣of␣Ontology', 'parent':'
        ↪ root', 'icd9_id':root_id},
    schema = ontology.schema,
)

ontology = pl.concat([
    ontology.filter(~pl.col('icd9_id').is_null()).sort('icd9_id'),
    uncoded,
    root_row,
])

dictionary = ontology.select(parent=pl.col('icd_code'), parent_id=pl.col('
    ↪ icd9_id'))
ontology = ontology.join(dictionary, how='left', on='parent')

# Save all the files

diagnoses_path  = os.path.join(output_prefix, output_diagnoses)
ccs_path        = os.path.join(output_prefix, output_ccs)
icd9_path       = os.path.join(output_prefix, output_icd)
ontology_path   = os.path.join(output_prefix, output_ontology)
generation_path = os.path.join(output_prefix, output_generation)

print('')
print(f'diagnoses␣path␣is:␣␣␣{diagnoses_path}')
print(f'ccs␣path␣is:␣␣␣␣␣␣␣␣{ccs_path}')
print(f'icd9␣path␣is:␣␣␣␣␣␣␣{icd9_path}')
print(f'ontology␣path␣is:␣␣␣{ontology_path}')
print(f'generative␣path␣is:␣{generation_path}')
print('')

diagnoses .write_parquet(diagnoses_path)
```

```
ccs_codes .write_parquet(ccs_path)
icd9_codes.write_parquet(icd9_path)
ontology  .write_parquet(ontology_path)
generation_conversion.write_parquet(generation_path)
```

This listing contains the definitions of the predictor and generator model we used with some other auxiliary function to use them.

```
import polars as pl
import numpy as np

import tomlkit

import torch
from torch import nn
import torch.nn.functional as F

import math
from dataclasses import dataclass
from typing import Self
import os

CFG_FILE_NAME   = 'config.toml' # this is the one that is reported in the
    ↪ save dir
MODEL_FILE_NAME = 'model.torch'

__all__ = [
    'Kelso_Filler', 'Kelso_Predictor', 'load_kelso_for_inference', '
        ↪ prepare_batch_for_inference',
    'Kelso_Config', 'load_kelso_for_generation', '
        ↪ prepare_batch_for_generation'
]

@dataclass(kw_only=True)
class Kelso_Config:
    vocab_size:  int
    output_size: int
    hidden_size: int
    num_layers:  int
    num_heads:   int
    head_dim:    int
    pos_base:    float
    dropout:     float
    device:      str | torch.device
    mlp_intermediate_size: int
    parametrized_head: bool


class Kelso_Model(nn.Module):
    def __init__(self, config: Kelso_Config, decoder_mask: bool):
        super().__init__()
        self.config = config
        self.decoder_mask = decoder_mask
        with torch.device(config.device):
            self.embedding = nn.Embedding(config.vocab_size, config.
                ↪ hidden_size)
            rotary_embedding = Rotary_Embedding (
                config.head_dim,
                config.pos_base,
                device = config.device,
            )
            self.decoder_layers = nn.ModuleList (
```

```
            [Kelso_Decoder_Layer(config, rotary_embedding) for _ in
                ↪ range(config.num_layers)]
        )
        self.pooling = Kelso_Pooling(config)
        self.head = nn.Linear(config.hidden_size, config.output_size,
            ↪ bias=True)

def forward (
    self,
    batch:     torch.Tensor,
    positions: torch.LongTensor,
    lengths:   torch.LongTensor,
    return_attention: bool = False,
) -> list[torch.Tensor]:
    # batch, position size is (bsz, b_n)
    if batch.shape != positions.shape:
        raise ValueError(
            f'position␣shape␣should␣be␣equal␣to␣batch␣shape.␣Expected:␣{
                ↪ batch.shape},␣'
            f'received:␣{positions.shape}'
        )
    # lenghts size is (bsz,)
    if batch.shape[0] != lengths.shape[0] or len(lengths.shape) >  1:
        raise ValueError(
            f'lengths␣shape␣should␣be␣({batch.shape[0]},)␣but␣it␣is␣{
                ↪ lengths.shape}'
        )

    bsz, b_n = positions.shape

    # PHASE 1: embed the code_ids

    # embedding has dim (bsz, n, hidden_size)
    embeddings = self.embedding(batch)

    # PHASE 2: create mask

    # attention mask has dim (bsz, 1, b_n, b_n)
    # decoder mask has dim (bsz, b_n, b_n)
    # pad mask has dim (bsz, b_n)

    minus_inf = torch.finfo(torch.float).min

    filter   = torch.arange(b_n, device=self.config.device).view((1, -1)
        ↪ )
    pad_mask = torch.full((bsz, b_n), minus_inf, device=self.config.
        ↪ device)
    pad_mask = pad_mask.masked_fill_(filter < lengths.view((-1, 1)), 0)

    pad_mask = pad_mask[:, None, None, :]

    if self.decoder_mask:
        decoder_mask = torch.full((bsz, b_n, b_n), 0.0, device=self.
            ↪ config.device)
        decoder_mask = decoder_mask.masked_fill_(positions.unsqueeze(2)
            ↪ < positions.unsqueeze(1), minus_inf)
        decoder_mask = decoder_mask[:, None, :, :]
        mask = pad_mask + decoder_mask
    else:
        mask = pad_mask

    # PHASE 3: transformer
```

```
        all_attentions = []

        batch = embeddings
        for layer in self.decoder_layers:
            batch, attention = layer(batch, positions, mask)
            all_attentions.append(attention)

        batch = F.dropout(batch, p=self.config.dropout, training=self.
            ↪ training)

        if return_attention:
            # this is of shape (n_layers, bsz, num_heads, b_n, b_n)
            all_attentions = torch.stack(all_attentions)
            # this is of shape (bsz, n_layers, num_heads, b_n, b_n)
            return batch, all_attentions.transpose(0, 1)
        else:
            return batch


class Kelso_Filler(nn.Module):
    def __init__(self, config: Kelso_Config):
        super().__init__()
        self.model  = Kelso_Model(config, decoder_mask=False)
        self.config = config
        with torch.device(config.device):
            self.head = nn.Linear(config.hidden_size, config.output_size,
                ↪ bias=True)

    def forward (
        self,
        batch:     torch.Tensor,
        positions: torch.LongTensor,
        lengths:   torch.LongTensor,
    ) -> list[torch.Tensor]:
        batch = self.model(batch, positions, lengths)
        result = self.head(batch)
        return result



class Kelso_Predictor(nn.Module):
    def __init__(self, config: Kelso_Config):
        super().__init__()
        self.config = config
        self.model  = Kelso_Model(config, decoder_mask=True)
        with torch.device(config.device):
            self.pooling = Kelso_Pooling(config)
            self.head    = nn.Linear(config.hidden_size, config.output_size,
                ↪  bias=True)

    def forward (
        self,
        batch:     torch.Tensor,
        positions: torch.LongTensor,
        lengths:   torch.LongTensor,
        return_attention: bool = False,
    ) -> list[torch.Tensor]:
        bsz = batch.shape[0]
        batch = self.model(batch, positions, lengths, return_attention)

        if return_attention:
```

```
            batch, attention = batch

        # this is of shape (bsz, b_m, hidden)
        batch_pooled = self.pooling(batch, positions, lengths)
        output = self.head(batch_pooled)

        result = []
        for it in range(bsz):
            t = output[it][:positions[it].max() + 1]
            result.append(t)
        if return_attention:
            return result, attention
        else:
            return result

# See this [article](http://arxiv.org/abs/2104.09864)
class Rotary_Embedding:
    dim:    int
    base:   float

    def __init__(self, dim: int, base: float, *, device):
        if dim % 2 != 0:
            raise ValueError (f'Tried␣to␣instanciate␣a␣rotary␣embedding␣with
                ↪ ␣a␣odd␣dimension␣({dim})')
        self.dim  = dim
        self.base = base

        self.max_seq_len = 0
        # @rubustness when we call the .to() method on the parent module,
            ↪ this should be moved too
        self.inv_freq = 1.0 / (self.base ** (torch.arange(0, self.dim, 2,
            ↪ device=device).float() / self.dim))

    def increase_cache(self, seq_len: int):
        self.max_seq_len = seq_len
        t     = torch.arange(seq_len, device=self.inv_freq.device)
        freqs = torch.einsum("i,j->ij", t, self.inv_freq)
        emb   = torch.cat((freqs, freqs), dim=-1)
        self.cos_buffer = emb.cos()
        self.sin_buffer = emb.sin()

    def __call__(self, batch: torch.Tensor, positions: torch.LongTensor) ->
        ↪ torch.Tensor:
        bsz, num_heads, b_n, head_dim = batch.shape
        if self.max_seq_len < b_n:
            self.increase_cache(max(b_n, 2*self.max_seq_len))
        batch_x = batch[..., : head_dim // 2  ]
        batch_y = batch[...,   head_dim // 2 :]
        rotated = torch.cat((-batch_y, batch_x), dim=-1)
        cos_f = self.cos_buffer[positions].unsqueeze(1) # we will broadcast
            ↪ over dim 1
        sin_f = self.sin_buffer[positions].unsqueeze(1) # we will broadcast
            ↪ over dim 1
        return batch*cos_f + rotated*sin_f


class Kelso_Decoder_Layer(nn.Module):
    def __init__(self, config: Kelso_Config, rotary_embedding:
        ↪ Rotary_Embedding):
        super().__init__()
        self.attention = Kelso_Attention(config, rotary_embedding)
        self.mlp = Kelso_MLP(config.hidden_size, config.
```

```
            ↪ mlp_intermediate_size)
        self.normalization_pre  = nn.LayerNorm(config.hidden_size)
        self.normalization_post = nn.LayerNorm(config.hidden_size)
        self.dropout = config.dropout

    def forward(self, batch: torch.Tensor, positions: torch.LongTensor, mask
        ↪ : torch.Tensor):
        residual = batch
        batch, attn = self.attention(batch, mask, positions)
        batch = F.dropout(batch, p=self.dropout, training=self.training)
        batch = batch + residual
        residual = batch
        batch = self.normalization_pre(batch)
        batch = self.mlp(batch)
        batch = F.dropout(batch, p=self.dropout, training=self.training)
        batch = batch + residual
        batch = self.normalization_post(batch)
        return batch, attn


class Kelso_Attention(nn.Module):
    def __init__(self, config: Kelso_Config, rotary_embedding:
        ↪ Rotary_Embedding|None = None):
        super().__init__()
        self.head_dim         = config.head_dim
        self.num_heads        = config.num_heads
        self.hidden_size      = config.hidden_size
        self.rotary_embedding = rotary_embedding

        self.q_proj = nn.Linear(self.hidden_size,               self.
            ↪ num_heads * self.head_dim, bias=False)
        self.k_proj = nn.Linear(self.hidden_size,               self.
            ↪ num_heads * self.head_dim, bias=False)
        self.v_proj = nn.Linear(self.hidden_size,               self.
            ↪ num_heads * self.head_dim, bias=False)
        self.o_proj = nn.Linear(self.num_heads * self.head_dim, self.
            ↪ hidden_size, bias=False)

        self.dropout = config.dropout

    def forward (
        self,
        hidden_states: torch.Tensor,
        mask:          torch.Tensor,
        positions:     torch.LongTensor|None = None,
    ):
        bsz, b_n, _ = hidden_states.shape

        query_states = self.q_proj(hidden_states)
        key_states   = self.k_proj(hidden_states)
        value_states = self.v_proj(hidden_states)
        query_states = query_states.view(bsz, b_n, self.num_heads, self.
            ↪ head_dim).transpose(1, 2)
        key_states   = key_states  .view(bsz, b_n, self.num_heads, self.
            ↪ head_dim).transpose(1, 2)
        value_states = value_states.view(bsz, b_n, self.num_heads, self.
            ↪ head_dim).transpose(1, 2)

        if self.rotary_embedding is not None:
            query_states = self.rotary_embedding(query_states, positions)
            key_states   = self.rotary_embedding(key_states,   positions)
        else:
```

```
            if positions is not None:
                raise ValueError('positions␣provided␣but␣no␣rotary␣embedding
                    ↪ ␣is␣set')

        attn_logits = torch.matmul(query_states, key_states.transpose(2, 3))
            ↪  / math.sqrt(self.head_dim)
        attn_logits = attn_logits + mask

        attn_weights = F.softmax(attn_logits, dim=-1)
        attn_output  = torch.matmul(attn_weights, value_states)

        if attn_output.size() != (bsz, self.num_heads, b_n, self.head_dim):
            raise ValueError(
                f"`attn_output`␣should␣be␣of␣size␣{(bsz,␣self.num_heads,␣b_n
                    ↪ ,␣self.head_dim)},␣but␣is"
                f"␣{attn_output.size()}"
            )

        attn_output = attn_output.transpose(1, 2).contiguous()
        attn_output = attn_output.view(bsz, b_n, self.num_heads * self.
            ↪ head_dim)
        attn_output = F.dropout(attn_output, p=self.dropout, training=self.
            ↪ training)
        attn_output = self.o_proj(attn_output)

        return attn_output, attn_weights

class Kelso_MLP(nn.Module):
    def __init__(self, hidden_size: int, intermediate_size: int):
        super().__init__()
        self.gate_proj = nn.Linear(hidden_size, intermediate_size, bias=
            ↪ False)
        self.up_proj   = nn.Linear(hidden_size, intermediate_size, bias=
            ↪ False)
        self.down_proj = nn.Linear(intermediate_size, hidden_size, bias=
            ↪ False)
        self.act_fn    = F.silu

    def forward(self, x):
        down_proj = self.down_proj(self.act_fn(self.gate_proj(x)) * self.
            ↪ up_proj(x))
        return down_proj

class Kelso_Pooling(nn.Module):
    def __init__(self, config: Kelso_Config):
        super().__init__()
        self.parametrized_head = config.parametrized_head
        if config.parametrized_head:
            self.scorer_attn = Kelso_Attention(config)
            self.scorer_gmlp = Kelso_MLP(config.hidden_size, config.
                ↪ mlp_intermediate_size)

    def forward(self, batch: torch.Tensor, pos: torch.LongTensor, lengths:
        ↪ torch.LongTensor) -> torch.Tensor:
        bsz, b_n, h = batch.shape
        b_m = pos.max() + 1
        minus_inf = torch.finfo(torch.float).min

        # compute pooling scores
        with torch.device(batch.device):
            decoder_mask = torch.full((bsz, b_n, b_n), 0.0)
            decoder_mask = decoder_mask.masked_fill_(pos.unsqueeze(2) != pos
```

```
                ↪ .unsqueeze(1), minus_inf)

            filter   = torch.arange(b_n).view((1, -1))
            pad_mask = torch.full((bsz, b_n), minus_inf)
            pad_mask = pad_mask.masked_fill_(filter < lengths.view((-1, 1)),
                ↪  0)

        decoder_mask = decoder_mask[:, None, :, :]
        pad_mask = pad_mask[:, None, None, :]
        mask = pad_mask + decoder_mask

        if self.parametrized_head:
            pooling_score = self.scorer_attn(batch, mask)
            pooling_score = self.scorer_gmlp(batch)
        else:
            with torch.device(batch.device):
                pooling_score = torch.ones((bsz, b_n, h))

        # visit-wise sums helpers
        ids = pos.clamp(0, None) + torch.arange(bsz, device=pos.device).
            ↪ unsqueeze(1) * b_m
        ids = ids.flatten()

        # Compute visit-wise softmax
        pooling_mask = torch.zeros((bsz, b_n, 1), device=batch.device)
        pooling_mask.masked_fill_((pos == -1).unsqueeze(2), minus_inf)

        pooling_score += pooling_mask
        pooling_score = torch.exp(pooling_score)
        pooling_score = pooling_score.reshape(-1, h)

        sums = torch.zeros((bsz * b_m, h), device=batch.device)
        sums = sums.index_add_(0, ids, pooling_score)
        pooling_probs = pooling_score / sums[ids]

        # now pool
        batch = batch.reshape((-1, h)) * pooling_probs
        result = torch.zeros((bsz * b_m, h), device=batch.device)
        result = result.index_add_(0, ids, batch)
        result = result.reshape((bsz, b_m, h))

        return result


@dataclass
class Inference_Batch:
    codes:     torch.Tensor
    positions: torch.Tensor
    lenghts:   torch.Tensor

    def unpack(self) -> dict:
        return {
            'batch':     self.codes,
            'positions': self.positions,
            'lengths':   self.lenghts,
        }

def prepare_batch_for_inference (
    codes:     list[np.ndarray],
    counts:    list[np.ndarray],
    positions: list[np.ndarray],
    device:    torch.device,
```

```
) -> Inference_Batch:
    codes     = [x.astype(np.int64) for x in codes]
    counts    = [x.astype(np.int64) for x in counts]
    positions = [x.astype(np.int64) for x in positions]

    lengths = [len(x) for x in codes]
    b_n = max(lengths)

    b_codes     = np.array([np.pad(x, (0, b_n - len(x)), constant_values=0 )
        ↪  for x in codes])
    b_positions = np.array([np.pad(x, (0, b_n - len(x)), constant_values=-1)
        ↪  for x in positions])

    with torch.device(device):
        b_codes     = torch.from_numpy(b_codes)    .to(device)
        b_positions = torch.from_numpy(b_positions).to(device)
        b_lengths   = torch.LongTensor(lengths)    .to(device)

    return Inference_Batch (
        codes     = b_codes,
        positions = b_positions,
        lenghts   = b_lengths,
    )

@dataclass
class Generation_Batch:
    codes:     torch.Tensor
    positions: torch.Tensor
    lenghts:   torch.Tensor

    def unpack(self) -> dict:
        return {
            'batch':     self.codes,
            'positions': self.positions,
            'lengths':   self.lenghts,
        }

def prepare_batch_for_generation (
    codes:     list[np.ndarray],
    counts:    list[np.ndarray],
    positions: list[np.ndarray],
    hole_prob: float,
    hole_token_id: int,
    device:    torch.device,
) -> Generation_Batch:
    codes     = [x.astype(np.int64) for x in codes]
    counts    = [x.astype(np.int64) for x in counts]
    positions = [x.astype(np.int64) for x in positions]

    lengths = [len(x) for x in codes]
    b_n = max(lengths)

    b_codes     = np.array([np.pad(x, (0, b_n - len(x)), constant_values=0 )
        ↪  for x in codes])
    b_positions = np.array([np.pad(x, (0, b_n - len(x)), constant_values=-1)
        ↪  for x in positions])

    mask = np.random.rand(*b_codes.shape) < hole_prob
    b_codes[mask] = hole_token_id

    with torch.device(device):
        b_codes     = torch.from_numpy(b_codes)    .to(device)
```

```
        b_positions = torch.from_numpy(b_positions).to(device)
        b_lengths   = torch.LongTensor(lengths)    .to(device)

    return Inference_Batch (
        codes     = b_codes,
        positions = b_positions,
        lenghts   = b_lengths,
    )



def load_kelso_for_inference(path: str) -> Kelso_Predictor:
    config_path = os.path.join(path, CFG_FILE_NAME)
    model_path  = os.path.join(path, MODEL_FILE_NAME)

    with open(config_path, 'r') as f:
        txt = f.read()
    config = tomlkit.parse(txt)['model']
    config = Kelso_Config(**config)

    model = Kelso_Predictor(config)
    state_dict = torch.load(model_path)
    model.load_state_dict(state_dict)
    return model

def load_kelso_for_generation(path: str) -> Kelso_Filler:
    config_path = os.path.join(path, CFG_FILE_NAME)
    model_path  = os.path.join(path, MODEL_FILE_NAME)

    with open(config_path, 'r') as f:
        txt = f.read()
    config = tomlkit.parse(txt)
    hole_prob     = config['trainer']['hole_prob']
    hole_token_id = config['trainer']['hole_token_id']
    config = Kelso_Config(**config['model'])

    model = Kelso_Filler(config)
    state_dict = torch.load(model_path)
    model.load_state_dict(state_dict)

    return model, hole_prob, hole_token_id
```

This code is used to train a predictor model. Similar code is used to perform the hyperparameter selection, but we omit it.

```
import math
import os
from dataclasses import dataclass

import polars as pl
import numpy as np
import random

import torch
from torch import nn
import torch.nn.functional as F

from tqdm import tqdm
import tomlkit
from datetime import datetime
```

```
from kelso_model import Kelso_Config, Kelso_Predictor, CFG_FILE_NAME,
    ↪ MODEL_FILE_NAME

CONFIG_FILE_PATH = 'repo/kelso/config.toml'

LOG_FILE_NAME = 'log.txt'
CSV_FILE_NAME = 'log.csv'

def nice_print(txt: str):
    tqdm.write(txt)

@dataclass(kw_only=True)
class Metrics_Config:
    recalls: list[int]

@dataclass(kw_only=True)
class Trainer_Config:
    batch_size:        int
    num_epochs:        int
    learning_rate:     float
    eval_split:        float
    test_split:        float
    weight_decay:      float
    eval_batch_size:   int
    ccs_as_inputs:     bool
    save_directory:    str
    patience:          int
    max_patient_len:   int
    limit_num_batches: int | None = None

    # auto-filled
    metrics_config: Metrics_Config | None = None

@dataclass(kw_only=True)
class Epoch_Metrics:
    epoch:      int
    learn_rate: float
    train_loss: float
    eval_loss:  float
    recalls:    dict[int, float]

def log_metrics(metrics: Epoch_Metrics, trainer_config: Trainer_Config):
    txt_file_path = os.path.join(trainer_config.save_directory,
        ↪ LOG_FILE_NAME)
    csv_file_path = os.path.join(trainer_config.save_directory,
        ↪ CSV_FILE_NAME)

    txt  = f'{metrics.epoch:␣>3}.␣'
    txt += f'train_loss:␣{metrics.train_loss:.3f}'
    txt += f'␣␣␣eval␣loss:␣{metrics.eval_loss:.3f}'
    for k in trainer_config.metrics_config.recalls:
        txt += f"␣␣␣␣r{k}:␣{metrics.recalls[k]*100:.2f}%"
    txt += f"␣␣␣lr:{metrics.learn_rate:.3e}"

    nice_print(txt)
    with open(txt_file_path, 'a') as f:
        f.write(txt + '\n')

    if metrics.epoch == 0:
        if os.path.exists(csv_file_path):
            raise Error(f'We␣are␣in␣epoch␣zero,␣but␣csv␣file␣{csv_file_path}
                ↪ ␣already␣exists')
```

```
        txt  = 'epoch,learn_rate,train_loss,eval_loss,'
        txt += ','.join([f'recall_{k}' for k in sorted(metrics.recalls)])
        txt += '\n'
    else:
        txt = ''

    values = [
        metrics.epoch,
        metrics.learn_rate,
        metrics.train_loss,
        metrics.eval_loss,
    ]
    values += [metrics.recalls[k] for k in sorted(metrics.recalls)]
    values  = [str(v) for v in values]
    txt += ','.join(values)
    txt += '\n'

    with open(csv_file_path, 'a') as f:
        f.write(txt)


@dataclass(kw_only=True)
class Early_Stopper_Result:
    is_best_round:  bool
    should_exit: bool

class Early_Stopper:
    best_result: float
    best_epoch:  int | None
    patience:    int
    min_is_better: bool

    def __init__(self, patience: int, *, min_is_better: bool):
        self.best_result = 0.0
        self.best_epoch  = None
        self.patience    = patience
        self.min_is_better = min_is_better

    def check(self, epoch: int, metric: float) -> Early_Stopper_Result:
        check = self.best_epoch is None or metric < self.best_result
        if check == self.min_is_better:
            # then we are doing good
            self.best_epoch  = epoch
            self.best_result = metric
            is_best_round  = True
            should_exit = False
        else:
            is_best_round  = False
            should_exit = epoch - self.best_epoch > self.patience
        return Early_Stopper_Result(is_best_round=is_best_round, should_exit
            ↪ =should_exit)

@dataclass(kw_only=True)
class Training_Results:
    num_epochs: int
    loss: int
    recalls: dict[int, float]


def train(model: nn.Module, diagnoses: pl.DataFrame, trainer_config:
    ↪ Trainer_Config) -> Training_Results:
    def split(data):
```

```
    ccs_codes = list(data['ccs_id']  .to_numpy())
    if trainer_config.ccs_as_inputs:
        input_codes = list(data['ccs_id'] .to_numpy())
    else:
        input_codes = list(data['icd9_id'] .to_numpy())
    positions = list(data['position'].to_numpy())
    counts    = list(data['count']   .to_numpy())
    return ccs_codes, input_codes, counts, positions

ccs_train, input_train, counts_train, positions_train = split(diagnoses.
    ↪ filter(pl.col('role') == 'train'))
ccs_eval,  input_eval,  counts_eval,  positions_eval  = split(diagnoses.
    ↪ filter(pl.col('role') == 'eval'))
ccs_test,  input_test,  counts_test,  positions_test  = split(diagnoses.
    ↪ filter(pl.col('role') == 'test'))

# find the number of batches
num_batches = len(ccs_train) // trainer_config.batch_size
if trainer_config.limit_num_batches is not None:
    l_num_batches = min(num_batches, trainer_config.limit_num_batches)
    nice_print(
        f'dataset␣would␣contain␣{num_batches}␣batches␣of␣{trainer_config
            ↪ .batch_size}'
        f'␣but␣we␣are␣limiting␣it␣to␣{l_num_batches}'
    )
    num_batches = l_num_batches
else:
    nice_print(f'dataset␣contains␣{num_batches}␣batches␣of␣{
        ↪ trainer_config.batch_size}')

model_save_path = os.path.join(trainer_config.save_directory,
    ↪ MODEL_FILE_NAME)

#configure the optimizer
optimizer = torch.optim.AdamW (
    model.parameters(),
    trainer_config.learning_rate,
    weight_decay = trainer_config.weight_decay,
)

stopper = Early_Stopper(trainer_config.patience, min_is_better=True)

# Train Loop

try:
    for epoch in tqdm(range(trainer_config.num_epochs), 'epoch', leave=
        ↪ False):
        total_train_loss   = 0
        train_loss_divisor = 0

        for batch_id in tqdm(range(num_batches), 'train', leave=False):
            batch_start = batch_id * trainer_config.batch_size
            batch_end   = batch_start + trainer_config.batch_size

            # get the right data for the batch
            i_ccs       = ccs_train       [batch_start: batch_end]
            i_input     = input_train     [batch_start: batch_end]
            i_positions = positions_train [batch_start: batch_end]
            i_counts    = counts_train    [batch_start: batch_end]

            b_codes, b_positions, b_lengths, outputs = prepare_data (
                i_ccs, i_input, i_positions, i_counts,
```

```
            )

            # feed-forward + backpropagation
            optimizer.zero_grad()
            model.train()
            predictions = model(b_codes, b_positions, b_lengths)
            total_loss  = compute_loss(predictions, outputs)
            divisor     = sum([x.shape[0] for x in predictions])
            loss        = total_loss / divisor
            loss.backward()

            optimizer.step()

            total_train_loss   += float(total_loss)
            train_loss_divisor += divisor

        train_loss = total_train_loss / train_loss_divisor
        metrics_dict = evaluate (
            model,
            ccs_eval,
            input_eval,
            positions_eval,
            counts_eval,
            trainer_config,
        )

        metrics = Epoch_Metrics (
            epoch      = epoch,
            learn_rate = float(optimizer.param_groups[0]['lr']),
            train_loss = train_loss,
            eval_loss  = metrics_dict['loss'],
            recalls    = metrics_dict['recalls']
        )
        log_metrics(metrics, trainer_config)

        # The `min_is_better` field is relevant in constructor!!
        stopper_result = stopper.check(epoch, metrics.eval_loss)

        if stopper_result.is_best_round:
            torch.save(model.state_dict(), model_save_path)

        if stopper_result.should_exit:
            nice_print('It␣seems␣we␣are␣done␣here...')
            break
except KeyboardInterrupt:
    nice_print('exiting␣loop...')


model.load_state_dict(torch.load(model_save_path))
metrics_dict = evaluate (
    model,
    ccs_test,
    input_test,
    positions_test,
    counts_test,
    trainer_config,
)
training_results = Training_Results (
    num_epochs = epoch,
    loss       = metrics_dict['loss'],
    recalls    = metrics_dict['recalls'],
)
```

```
    return training_results


def evaluate (
    model: nn.Module,
    ccs_codes: list[np.ndarray],
    icd_codes: list[np.ndarray],
    positions: list[np.ndarray],
    counts:    list[np.ndarray],
    config:    Trainer_Config,
):
    # prepare the metrics dict
    metrics = {}
    metrics['loss'] = 0
    metrics['recalls'] = {}
    for k in config.metrics_config.recalls:
        metrics['recalls'][k] = 0
    divisor = 0

    num_batches = len(ccs_codes) // config.eval_batch_size
    for batch_id in tqdm(range(num_batches), ' eval', leave=False):
        batch_start = batch_id * config.eval_batch_size
        batch_end   = batch_start + config.eval_batch_size

        # get the right data for the batch
        i_icd       = icd_codes [batch_start: batch_end]
        i_ccs       = ccs_codes [batch_start: batch_end]
        i_positions = positions [batch_start: batch_end]
        i_counts    = counts    [batch_start: batch_end]

        b_codes, b_positions, b_lengths, outputs = prepare_data (
            i_ccs, i_icd, i_positions, i_counts,
        )

        # computations
        model.eval()
        with torch.inference_mode():
            predictions = model(b_codes, b_positions, b_lengths)
            m = compute_metrics(predictions, outputs, config.metrics_config)
        metrics['loss'] += m['loss']
        for k in config.metrics_config.recalls:
            metrics['recalls'][k] += m['recalls'][k]
        divisor += sum([x.shape[0] for x in predictions])

    metrics['loss'] /= divisor
    for k in config.metrics_config.recalls:
        metrics['recalls'][k] /= num_batches
    return metrics

def compute_loss(predictions, outputs) -> torch.Tensor:
    losses = []
    for pred, out in zip(predictions, outputs):
        # if reduce='none' the function returns the same shape as input
        loss = F.binary_cross_entropy_with_logits(pred, out, reduction='sum'
            ↪ )
        losses.append(loss)
    total_loss = sum(losses)
    return total_loss


def compute_metrics (predictions: list[torch.Tensor], outputs: list[torch.
    ↪ Tensor], config: Metrics_Config):
```

```
    metrics = {}
    metrics['loss'] = float(compute_loss(predictions, outputs))
    metrics['recalls'] = {}
    for k in config.recalls:
        rec = []
        for pred, out in zip(predictions, outputs):
            # create shifter to index the flatten array
            t = torch.ones(out.shape[0], dtype=int, device=out.device) * out
                ↪ .shape[-1]
            t[0] = 0
            t = t.cumsum(0).unsqueeze(1)

            sel = pred.topk(k, dim=-1).indices
            tp = out.flatten()[sel+t].sum(-1).to(float) # true positives
            tt = out.sum(-1)
            recall = (tp / tt).mean()
            rec.append(float(recall))
        metrics['recalls'][k] = sum(rec) / len(rec)
    return metrics


def prepare_data(i_ccs, i_input, i_positions, i_counts):
    # prepare the data for the model
    b_input     = [x[:-int(c[-1])]                  for x, c in zip(i_input,
        ↪     i_counts)]
    b_positions = [x[:-int(c[-1])].astype(np.int_) for x, c in zip(
        ↪ i_positions, i_counts)]
    lengths = [len(x) for x in b_input]
    b_n     = max(lengths)
    b_input     = np.array([np.pad(x, (0, b_n - len(x)), constant_values=0 )
        ↪  for x in b_input]).astype(np.int64)
    b_positions = np.array([np.pad(x, (0, b_n - len(x)), constant_values=-1)
        ↪  for x in b_positions])

    b_input     = torch.from_numpy(b_input)    .to(model.config.device)
    b_positions = torch.from_numpy(b_positions).to(model.config.device)
    b_lengths   = torch.LongTensor(lengths)    .to(model.config.device)

    # compute expected outputs for the loss
    outputs = []
    for it in range(b_input.shape[0]): # this is range(batch_size)
        sz = len(i_counts[it])
        out = np.zeros((sz-1, model.config.output_size), dtype=float)
        cursor = i_counts[it][0]
        for jt in range(1, sz):
            cursor_end = cursor + i_counts[it][jt]
            out[jt-1, i_ccs[it][cursor:cursor_end]] = 1
            cursor = cursor_end
        out = torch.from_numpy(out).to(torch.float).to(model.config.device)
        outputs.append(out)
    return b_input, b_positions, b_lengths, outputs

DIR_ID_LENGTH = 5
ALL_LETTERS   = 'abcdefghijklmnopqrstuvxywz'
def format_path(path: str) -> str:
    date = datetime.now().strftime("%Y-%m-%d_%H:%M:%S")
    id = ''.join([random.choice(ALL_LETTERS) for _ in range(DIR_ID_LENGTH)])
    path = path.replace('%(id)',   id)
    path = path.replace('%(date)', date)
    return path
```

```
if __name__ == '__main__':
    nice_print(f'Using␣config␣file:␣{CONFIG_FILE_PATH}')
    with open(CONFIG_FILE_PATH, 'r') as f:
        txt = f.read()
    config = tomlkit.parse(txt)

    diagnoses = pl.read_parquet(config['diagnoses_path'])

    config['model']['vocab_size']  = diagnoses['icd9_id'].list.max().max() +
        ↪  1
    config['model']['output_size'] = diagnoses['ccs_id'] .list.max().max() +
        ↪  1
    kelso_config = Kelso_Config(**config['model'])
    kelso_config.device = torch.device(kelso_config.device)
    model = Kelso_Predictor(kelso_config)

    num_params  = sum([param.nelement()                       for param in
        ↪ model.parameters()])
    size_params = sum([param.nelement()*param.element_size() for param in
        ↪ model.parameters()])
    nice_print(f'model␣has␣{num_params/1e6:.2f}M␣params,␣occupying␣{
        ↪ size_params/1024/1024:.2f}M␣of␣memory')

    trainer_config = Trainer_Config(**config['trainer'])
    metrics_config = Metrics_Config(**config['metrics'])
    trainer_config.metrics_config = metrics_config

    trainer_config.save_directory = format_path(trainer_config.
        ↪ save_directory)
    nice_print(f'>␣Save␣directory␣is␣{trainer_config.save_directory}')
    os.makedirs(trainer_config.save_directory)

    num_patients_before = diagnoses.shape[0]
    diagnoses = diagnoses.filter(pl.col('count').list.sum() < trainer_config
        ↪ .max_patient_len)
    # @debug
    # diagnoses = diagnoses.filter(pl.col('count').list.lengths() > 2)
    num_patients_after = diagnoses.shape[0]
    nice_print (
        f'Original␣dataset␣contains␣{num_patients_before}␣patients.␣We␣cut␣'
        f'those␣with␣more␣than␣{trainer_config.max_patient_len}␣codes,␣so␣we
            ↪ ␣are␣'
        f'left␣with␣{num_patients_after}␣patients␣'
        f'({(1␣-␣num_patients_after␣/␣num_patients_before)*100:.1f}%␣less).'
    )

    training_infos = tomlkit.table()
    training_infos['num_patients'] = num_patients_after
    training_infos['num_parameters'] = num_params
    config['training_infos'] = training_infos

    # update toml document
    new_config_text = tomlkit.dumps(config)
    new_config_path = os.path.join(trainer_config.save_directory,
        ↪ CFG_FILE_NAME)
    with open(new_config_path, 'w') as f:
        f.write(new_config_text)

    starting_time = datetime.now()

    results = train (
        model          = model,
```

```
        diagnoses      = diagnoses,
        trainer_config = trainer_config,
    )

    # compute training time
    end_time = datetime.now()
    time_delta = end_time - starting_time
    config['training_infos']['training_time'] = str(time_delta)

    # add test data to toml document
    test_results = tomlkit.table()
    test_results['training_epochs'] = results.num_epochs
    test_results['loss'] = results.loss
    for k in sorted(metrics_config.recalls):
        test_results[f'recall_{k}'] = results.recalls[k]
    config['test_results'] = test_results

    # save new results on config file
    new_config_text = tomlkit.dumps(config)
    with open(new_config_path, 'w') as f:
        f.write(new_config_text)

    # print the test results on screen
    txt = [f'test␣loss:␣{results.loss:.3f}']
    for k in sorted(metrics_config.recalls):
        txt += [f'recall_{k:␣<2}:␣{results.recalls[k]*100:.2f}%']
    txt = '␣␣␣␣'.join(txt)
    nice_print(txt)
```

The following code is used to make the prediction and the relative explanation. Similar code is used to generate the fidelity and the correlations with attention. It leverages on a low-level library on the most performance-intensive operations.

```
import polars as pl
import numpy as np

import lib.generator as gen

from kelso_model import *
import tomlkit

import torch
from sklearn.tree import DecisionTreeClassifier
from sklearn import metrics

import random

from tqdm import tqdm

ontology_path   = 'data/processed/ontology.parquet'
diagnoses_path  = 'data/processed/diagnoses.parquet'
generation_path = 'data/processed/generation.parquet'
ccs_path        = 'data/processed/ccs.parquet'
icd_path        = 'data/processed/icd.parquet'

config_path = 'repo/kelso/explain_config.toml'
output_path = 'results/explainer.txt'

model_path      = 'results/kelso2-dejlv-2024-01-20_17:23:33/'
filler_path     = 'results/filler-xyxdp-2024-01-21_15:16:51/'
```

```
k_reals           = 50
batch_size        = 64
keep_prob         = 0.8 # for ontological perturbation
topk_predictions = 10
ontological_perturbation   = True
generative_perturbation    = True
uniform_perturbation       = False
tree_train_fraction        = 0.75
synthetic_multiply_factor  = 4
generative_multiply_factor = 4

reference_index = 0 # which patient of the dataset to explain

def print_patient(ids: np.ndarray, cnt: np.ndarray, ontology: pl.DataFrame):
    codes = pl.DataFrame({'icd9_id':ids})
    codes = codes.join(ontology, how='left', on='icd9_id')

    cursor = 0
    for it in range(cnt.shape[0]):
        length = cnt[it]
        lines = []
        for jt in range(cursor, cursor+length):
            x = f'[{codes["icd_code"][jt]:}]'
            txt  = f'␣␣␣␣{x:␣<10}'
            txt += f'{codes["label"][jt]}'
            lines.append(txt)
        txt = '\n'.join(lines)
        print(f'visit␣{it+1}')
        print(txt)
        cursor += length

def explain_label(neigh_ccs, neigh_counts, labels, max_ccs_id,
    ↪ tree_train_fraction, target):
    # Tree fitting
    tree_inputs = gen.ids_to_encoded(neigh_ccs, neigh_counts, max_ccs_id,
        ↪ 0.5)

    tree_classifier = DecisionTreeClassifier()

    train_split = int(tree_train_fraction * len(labels))

    tree_inputs_train = tree_inputs[:train_split]
    tree_inputs_eval  = tree_inputs[train_split:]
    labels_train      = labels[:train_split, target]
    labels_eval       = labels[train_split:, target]

    tree_classifier.fit(tree_inputs_train, labels_train)

    return tree_classifier

# Load base data and model

ontology  = pl.read_parquet(ontology_path)
diagnoses = pl.read_parquet(diagnoses_path)
ccs_data  = pl.read_parquet(ccs_path)

diagnoses_train = diagnoses.filter(pl.col('role') == 'train')
diagnoses_eval  = diagnoses.filter(pl.col('role') == 'eval')

unique_codes = diagnoses['icd9_id'].explode().unique().to_numpy()
max_ccs_id = ccs_data['ccs_id'].max() + 1
```

```
ontology_array = ontology[['icd9_id', 'parent_id']].to_numpy()
gen.create_c2c_table(ontology_array, unique_codes)


# Extract the numpy data

ccs_codes_train = list(diagnoses_train['ccs_id'  ].to_numpy())
icd_codes_train = list(diagnoses_train['icd9_id' ].to_numpy())
positions_train = list(diagnoses_train['position'].to_numpy())
counts_train    = list(diagnoses_train['count'   ].to_numpy())

ccs_codes_eval = list(diagnoses_eval['ccs_id'  ].to_numpy())
icd_codes_eval = list(diagnoses_eval['icd9_id' ].to_numpy())
positions_eval = list(diagnoses_eval['position'].to_numpy())
counts_eval    = list(diagnoses_eval['count'   ].to_numpy())


model = load_kelso_for_inference(model_path)

total_accuracy = 0.0
total_f1_score = 0.0

num_layers = len(model.model.decoder_layers)
importance_analysis = ['importance', 'max', 'min', 'avg']
importance_source = ['all'] + ['layer_{}'.format(x) for x in range(
    ↪ num_layers)]
correlation_matrices = {s: np.zeros((4,4)) for s in importance_source}


if generative_perturbation:
    filler, hole_prob, hole_token_id = load_kelso_for_generation(filler_path
        ↪ )
    conv_data = pl.read_parquet(generation_path).sort('out_id')
    zero_row  = pl.DataFrame({'icd9_id':0, 'out_id':0, 'ccs_id':0}, schema=
        ↪ conv_data.schema)
    conv_data = pl.concat([zero_row, conv_data])

    out_to_icd = conv_data['icd9_id'].to_numpy()
    out_to_ccs = conv_data['ccs_id' ].to_numpy()

# Find closest neighbours in the real data
distance_list = gen.compute_patients_distances (
    icd_codes_eval[reference_index],
    counts_eval[reference_index],
    icd_codes_train,
    counts_train,
)
topk = np.argpartition(distance_list, k_reals-1)[:k_reals]

neigh_icd       = []
neigh_ccs       = []
neigh_counts    = []
neigh_positions = []
for it in range(k_reals):
    neigh_icd      .append(icd_codes_train[topk[it]])
    neigh_ccs      .append(ccs_codes_train[topk[it]])
    neigh_counts   .append(counts_train   [topk[it]])
    neigh_positions.append(positions_train[topk[it]])

neigh_icd      .append(icd_codes_eval[reference_index])
neigh_ccs      .append(ccs_codes_eval[reference_index])
neigh_counts   .append(counts_eval   [reference_index])
```

```
neigh_positions.append(positions_eval[reference_index])


# augment the neighbours with some synthetic points

if ontological_perturbation:
    displacements, new_counts = gen.ontological_perturbation(neigh_icd,
        ↪ neigh_counts, synthetic_multiply_factor, keep_prob)

    new_neigh_icd       = []
    new_neigh_ccs       = []
    new_neigh_positions = []
    for it, (icd, ccs, pos) in enumerate(zip(neigh_icd, neigh_ccs,
        ↪ neigh_positions)):
        for jt in range(synthetic_multiply_factor):
            displ = displacements[synthetic_multiply_factor * it + jt]
            new_neigh_icd      .append(icd[displ])
            new_neigh_ccs      .append(ccs[displ])
            new_neigh_positions.append(pos[displ])
    neigh_icd       += new_neigh_icd
    neigh_ccs       += new_neigh_ccs
    neigh_positions += new_neigh_positions
    neigh_counts += new_counts

if generative_perturbation:
    new_neigh_icd       = []
    new_neigh_ccs       = []
    new_neigh_counts    = []
    new_neigh_positions = []
    cursor = 0
    while cursor < len(neigh_icd):
        new_cursor = min(cursor + batch_size, len(neigh_icd))

        for _ in range(generative_multiply_factor):
            batch = prepare_batch_for_generation (
                neigh_icd      [cursor:new_cursor],
                neigh_counts   [cursor:new_cursor],
                neigh_positions[cursor:new_cursor],
                hole_prob,
                hole_token_id,
                torch.device('cuda')
            )

            if uniform_perturbation:
                bsz = batch.codes.shape[0]
                b_n = batch.codes.shape[1]
                n_out = filler.head.out_features
                gen_output = torch.zeros((bsz, b_n, n_out))
            else:
                gen_output = filler(**batch.unpack()) # (bsz, b_n, n_out)
            old_shape = gen_output.shape
            gen_output = gen_output.reshape((-1, gen_output.shape[-1]))
            gen_output = torch.softmax(gen_output, dim=-1)

            new_codes = torch.multinomial(gen_output, 1)
            new_codes = new_codes.reshape(old_shape[:-1])
            new_codes = new_codes.cpu().numpy()

            new_icd = list(out_to_icd[new_codes])
            new_ccs = list(out_to_ccs[new_codes])

            new_neigh_icd       += new_icd
```

```
            new_neigh_ccs       += new_ccs
            new_neigh_counts    += neigh_counts[cursor:new_cursor]
            new_neigh_positions += neigh_positions[cursor:new_cursor]

        cursor = new_cursor
    neigh_icd       += new_neigh_icd
    neigh_ccs       += new_neigh_ccs
    neigh_counts    += new_neigh_counts
    neigh_positions += new_neigh_positions


# Black Box Predictions on the reference

batch = prepare_batch_for_inference(
    [icd_codes_eval[reference_index]],
    [counts_eval[reference_index]],
    [positions_eval[reference_index]],
    torch.device('cuda'),
)
# attentions has shape (1, num_layers, num_heads, b_n, b_n)
output, attention = model(**batch.unpack(), return_attention=True)
output = output[0][-1]
labels = output.topk(topk_predictions).indices

arr_present_ccs = ccs_codes_eval[reference_index]
present_ccs = np.zeros((max_ccs_id,), dtype=bool)
for i in range(len(arr_present_ccs)):
    present_ccs[arr_present_ccs[i]] = True

attention = attention.squeeze(0)
attention_all = analyze_attention(attention, reference_index)
attention_layers = [analyze_attention(attention[x], reference_index) for x
    ↪ in range(attention.shape[0])]

# Black Box Predictions on neighbours

neigh_labels = np.empty((len(labels), len(neigh_icd), ), dtype=np.bool_)
cursor = 0
while cursor < len(neigh_icd):
    new_cursor = min(cursor+batch_size, len(neigh_icd))
    batch   = prepare_batch_for_inference (
        neigh_icd      [cursor:new_cursor],
        neigh_counts   [cursor:new_cursor],
        neigh_positions[cursor:new_cursor],
        torch.device('cuda')
    )
    outputs = model(**batch.unpack())

    outputs = [x[-1] for x in outputs]
    outputs = torch.stack(outputs)
    batch_labels = outputs.topk(topk_predictions, dim=-1).indices
    batch_labels = (batch_labels == labels[:,None,None]).any(-1)
    batch_labels = batch_labels.cpu().numpy()

    neigh_labels[:, cursor:new_cursor] = batch_labels
    cursor = new_cursor
neigh_labels = neigh_labels.transpose()

# explanation

tree = explain_label (
    neigh_ccs, neigh_counts, neigh_labels, max_ccs_id, tree_train_fraction,
```

```
        ↪ list(range(topk_predictions))
)

reference_enc = gen.ids_to_encoded(
    [ccs_codes_eval[reference_index]],
    [counts_eval[reference_index]],
    max_ccs_id,
    0.5
)[0]


# extract rules from explanation

tree_path = tree.tree_.decision_path(reference_enc.reshape((1,-1))).indices
features = tree.tree_.feature

expl_labels = [features[i] for i in tree_path]
expl_labels = [x for x in expl_labels if x >= 0]

thresholds = tree.tree_.threshold
thresholds = [thresholds[i] for i in tree_path if features[i] >= 0]

df = pl.DataFrame({'ccs_id': expl_labels, 'threasholds': thresholds}).
    ↪ with_columns(ccs_id=pl.col('ccs_id').cast(pl.UInt32))
df = df.join(ccs_data, on='ccs_id', how='left')

print_patient(icd_codes_eval[reference_index], counts_eval[reference_index],
    ↪  ontology)
print('\n')

print('ccs␣predicted')
labels = labels.tolist()
for id in labels:
    infos = ccs_data.filter(pl.col('ccs_id') == id)
    if len(infos) != 1:
        raise ValueError('should␣not␣happen')

    code = infos['ccs'][0]
    label = infos['description'][0]

    print(f'{"["+str(code)+"]":␣<7}␣{label}')


print('\ndecision␣rules')
for (id, thresh, ccs, desc) in df.iter_rows():
    txt = f'code␣{ccs:␣<4}␣-␣threashold:␣{thresh:.2f},␣found␣{reference_enc[
        ↪ id]:.2f},␣␣␣({desc})'
    print(txt)

print('\ncodes␣of␣the␣patient␣relevant␣for␣explanation')
for id in expl_labels:
    for it in range(len(ccs_codes_eval[reference_index])):
        ccs = ccs_codes_eval[reference_index][it]
        if ccs == id:
            visit = positions_eval[reference_index][it] + 1
            icd   = icd_codes_eval[reference_index][it] + 1

            infos = ontology.filter(pl.col('icd9_id') == icd)
            if len(infos) != 1:
                raise ValueError('Should␣not␣happen')
            code = infos['icd_code'][0]
            label = infos['label'][0]
```

```
            print(f'At visit {(str(visit)+","): <3} code {code: <6} [{label
                ↪ }]')
```

This Zig code contains most of the low-level library used in the explanation.

```
// This is chosen looking at the input table_c2c.
// Change this only if AoB errors occur (with the current ontology it doesnt
    ↪ )
const genealogy_max_size: usize = 12;

// number of parallel processes
const num_jobs: usize = 16;

// the numpy bindings
var np: Np = undefined;

// Memory management
var _arena: std.heap.ArenaAllocator = undefined;
var global_arena: std.mem.Allocator = undefined;

// Python object to report errors. Refer to pyhton documentation
var generator_error: ?*py.PyObject = null;

// This python module will export three functions
// Refer to the C bindings documentation of python for the format
//
// Functions referred to by this structure must be marked as `export` in
    ↪ their signature
const module_methods = [_]py.PyMethodDef{
    .{
        .ml_name = "create_c2c_table",
        .ml_meth = create_c2c_table,
        .ml_flags = py.METH_VARARGS,
        .ml_doc = "Precompute the code2code distances",
    },
    .{
        .ml_name = "compute_patients_distances",
        .ml_meth = _compute_patients_distances,
        .ml_flags = py.METH_VARARGS,
        .ml_doc = "inputs a patient (ids + count), a list of patients (list
            ↪ of ids + list of counts) and a neighborhood size",
    },
    .{
        .ml_name = "ids_to_encoded",
        .ml_meth = _ids_to_encoded,
        .ml_flags = py.METH_VARARGS,
        .ml_doc = "Perform the temporal encoding of doctorXAI",
    },
    .{
        .ml_name = "independent_perturbation",
        .ml_meth = _independent_perturbation,
        .ml_flags = py.METH_VARARGS,
        .ml_doc = "Simple perturbation method. Unused",
    },
    .{
        .ml_name = "ontological_perturbation",
        .ml_meth = _ontological_perturbation,
        .ml_flags = py.METH_VARARGS,
        .ml_doc = "Ontological perturbation of doctorXAI",
    },
```

```
    // this one is just a sentinel
    // It is an array entry with all values set to zero
    std.mem.zeroes(py.PyMethodDef),
};

// precomputed table of code2code distances
var table_c2c: Table_c2c = undefined;
// Ontology
var ontology: []u32 = undefined;

/// inputs a patient (ids + count), a list of patients (list of ids + list
    ↪ of counts)
/// returns an an array of distances of each element of the list with the
    ↪ input patient
/// call this only after having filled the c2c table (called `
    ↪ create_c2c_table` from python)
export fn _compute_patients_distances(self_obj: ?*py.PyObject, args: ?*py.
    ↪ PyObject) ?*py.PyObject {
    _ = self_obj;
    var arg1: ?*py.PyObject = undefined;
    var arg2: ?*py.PyObject = undefined;
    var arg3: ?*py.PyObject = undefined;
    var arg4: ?*py.PyObject = undefined;

    if (py.PyArg_ParseTuple(args, "OOOO", &arg1, &arg2, &arg3, &arg4) == 0)
        ↪ return null;

    const patient_codes = arg1;
    const patient_count = arg2;
    const dataset_codes = arg3;
    const dataset_count = arg4;

    var arena = std.heap.ArenaAllocator.init(std.heap.page_allocator);
    defer arena.deinit();
    const allocator = arena.allocator();

    const patient = parse_patient(patient_codes, patient_count, allocator)
        ↪ orelse {
        py.PyErr_SetString(generator_error, "Error while parsing patient");
        return null;
    };

    const dataset = parse_list_of_patients(dataset_codes, dataset_count,
        ↪ allocator) orelse return null;

    var result_data: []f32 = undefined;
    const result_array = blk: {
        var dimensions = [_]isize{@intCast(dataset.len)};
        var obj = np.simple_new(dimensions.len, &dimensions, Np.Types.FLOAT)
            ↪  orelse {
            py.PyErr_SetString(generator_error, "Failed while creating
                ↪ result array");
            return null;
        };
        var arr: *Np.Array_Obj = @ptrCast(obj);
        result_data = @as([*]f32, @ptrCast(@alignCast(arr.data)))[0..dataset
            ↪ .len];
        break :blk obj;
    };

    compute_patients_distances(patient, dataset, table_c2c, result_data);
```

```
    return result_array;
}

/// This takes the list of patients to encode (ids and counts) and the max
    ↪ id, then returns their encoded form
/// with temporal encoding
export fn _ids_to_encoded(self_obj: ?*py.PyObject, args: ?*py.PyObject) ?*py
    ↪ .PyObject {
    _ = self_obj;
    var arg1: ?*py.PyObject = undefined;
    var arg2: ?*py.PyObject = undefined;
    var arg3: i64 = undefined;
    var arg4: f32 = undefined;

    if (py.PyArg_ParseTuple(args, "OOlf", &arg1, &arg2, &arg3, &arg4) == 0)
        ↪ return null;

    const max_id: usize = @intCast(arg3);
    const lambda: f32 = arg4;

    var arena = std.heap.ArenaAllocator.init(std.heap.page_allocator);
    defer arena.deinit();
    const allocator = arena.allocator();

    const dataset = parse_list_of_patients(arg1, arg2, allocator) orelse
        ↪ return null;

    var encoded_data: [*]f32 = undefined;
    const encoded_array = blk: {
        var dimensions = [_]isize{ @intCast(dataset.len), @intCast(max_id)
            ↪ };
        var obj = np.simple_new(dimensions.len, &dimensions, Np.Types.FLOAT)
            ↪  orelse {
            py.PyErr_SetString(generator_error, "Failed while creating `
                ↪ encoded` result array");
            return null;
        };
        var arr: *Np.Array_Obj = @ptrCast(obj);
        encoded_data = @ptrCast(@alignCast(arr.data));
        break :blk obj;
    };

    ids_to_encoded(dataset, encoded_data, @intCast(max_id), lambda);

    return encoded_array;
}

export fn _independent_perturbation(self_obj: ?*py.PyObject, args: ?*py.
    ↪ PyObject) ?*py.PyObject {
    _ = self_obj;
    var arg1: ?*py.PyObject = undefined;
    var arg2: ?*py.PyObject = undefined;
    var arg3: c_int = undefined;
    var arg4: f32 = undefined;

    if (py.PyArg_ParseTuple(args, "OOlf", &arg1, &arg2, &arg3, &arg4) == 0)
        ↪ return null;

    const keep_prob: f32 = arg4;
    const multiply_factor: usize = @intCast(arg3);

    var arena = std.heap.ArenaAllocator.init(std.heap.page_allocator);
```

```
    defer arena.deinit();
    const allocator = arena.allocator();

    const dataset = parse_list_of_patients(arg1, arg2, allocator) orelse
        ↪ return null;
    const result = independent_perturbation(dataset, multiply_factor,
        ↪ keep_prob);

    return result;
}

export fn _ontological_perturbation(self_obj: ?*py.PyObject, args: ?*py.
    ↪ PyObject) ?*py.PyObject {
    _ = self_obj;
    var arg1: ?*py.PyObject = undefined;
    var arg2: ?*py.PyObject = undefined;
    var arg3: c_int = undefined;
    var arg4: f32 = undefined;

    if (py.PyArg_ParseTuple(args, "OOlf", &arg1, &arg2, &arg3, &arg4) == 0)
        ↪ return null;

    const keep_prob: f32 = arg4;
    const multiply_factor: usize = @intCast(arg3);

    var arena = std.heap.ArenaAllocator.init(std.heap.page_allocator);
    defer arena.deinit();
    const allocator = arena.allocator();

    const dataset = parse_list_of_patients(arg1, arg2, allocator) orelse
        ↪ return null;
    const result = ontological_perturbation(dataset, multiply_factor,
        ↪ keep_prob, ontology);

    return result;
}
const Table_c2c = struct {
    table: []f32,
    w: usize,

    inline fn get(self: Table_c2c, c1: u32, c2: u32) f32 {
        return self.table[c1 * self.w + c2];
    }

    inline fn get_mut(self: Table_c2c, c1: u32, c2: u32) *f32 {
        return &self.table[c1 * self.w + c2];
    }
};

/// Precomputes the code2code distances
export fn create_c2c_table(self_obj: ?*py.PyObject, args: ?*py.PyObject) ?*
    ↪ py.PyObject {
    // Unwrap arguments to a plain array
    _ = self_obj;
    var arg1: ?*py.PyObject = undefined;
    var arg2: ?*py.PyObject = undefined;

    if (py.PyArg_ParseTuple(args, "OO", &arg1, &arg2) == 0) return null;
    const _arr = np.from_otf(arg1, Np.Types.UINT, Np.Array_Flags.IN_ARRAY)
        ↪ orelse return null;
    const arr: *Np.Array_Obj = @ptrCast(_arr);
```

```
const _arr2 = np.from_otf(arg2, Np.Types.UINT, Np.Array_Flags.IN_ARRAY)
    ↪ orelse return null;
const arr2: *Np.Array_Obj = @ptrCast(_arr2);

var size: usize = undefined;
var tree: []u32 = undefined;
{
    const data: [*]u32 = @ptrCast(@alignCast(arr.data));
    const nd = arr.nd;
    if (nd != 2) {
        py.PyErr_SetString(generator_error, "Array should have a single
            ↪ dimension");
        return null;
    }
    const num_fields: usize = @intCast(arr.dimensions[1]);
    if (num_fields != 2) {
        const message = std.fmt.allocPrint(
            global_arena,
            "Array should have dim (*, 2), found dim ({}, {})",
            .{ arr.dimensions[0], arr.dimensions[1] },
        ) catch "Error";
        py.PyErr_SetString(generator_error, @ptrCast(message));
        return null;
    }
    size = @intCast(arr.dimensions[0]);
    tree = data[0 .. 2 * size];
}

const leaf_indices: []u32 = blk: {
    const t: [*]u32 = @ptrCast(@alignCast(arr2.data));
    const _size: usize = @intCast(arr2.dimensions[0]);
    break :blk t[0.._size];
};
// end of unwrapping
//
const max_leaf_index = std.mem.max(u32, leaf_indices) + 1;
const max_id = blk: {
    var max: u32 = 0;
    for (0..size) |it| {
        max = @max(tree[2 * it], max);
    }
    break :blk max;
};

// create and fill the ontology in a tree form
var _ontology_tree = global_arena.alloc(u32, max_id + 1) catch return
    ↪ null;
for (0..size) |it| {
    const child = tree[2 * it];
    const parent = tree[2 * it + 1];
    _ontology_tree[child] = parent;
}

// create and fill c2c table
table_c2c = std.mem.zeroes(Table_c2c);
table_c2c.table = global_arena.alloc(f32, max_leaf_index *
    ↪ max_leaf_index) catch @panic("Alloc error");
table_c2c.w = max_leaf_index;
@memset(table_c2c.table, std.math.nan(f32));

// this if distinguishes between the single-threaded and the multi-
    ↪ threaded implementations
```

```
// the result on the two branches should be the same
if (comptime num_jobs == 1) {
    for (leaf_indices, 0..) |it, it_index| {
        for (leaf_indices[0 .. it_index + 1]) |jt| {
            const dist = compute_c2c(it, jt, _ontology_tree);
            table_c2c.get_mut(it, jt).* = dist; // method in the
                ↪ Table_c2c struct
            table_c2c.get_mut(jt, it).* = dist; // method in the
                ↪ Table_c2c struct
        }
    }
} else {
    const Thread_Data = struct {
        ontology_tree: []u32,
        leaf_indices: []u32,
        table_c2c: Table_c2c,
        start: usize,
        len: usize,

        const Self = @This();

        pub fn job(self: Self) void {
            for (self.leaf_indices[self.start .. self.start + self.len],
                ↪  0..) |it, it_index| {
                for (self.leaf_indices[0 .. self.start + it_index + 1])
                    ↪ |jt| {
                    const dist = compute_c2c(it, jt, self.ontology_tree)
                        ↪ ;
                    table_c2c.get_mut(it, jt).* = dist;
                    table_c2c.get_mut(jt, it).* = dist;
                }
            }
        }
    };

    var arena = std.heap.ArenaAllocator.init(std.heap.page_allocator);
    defer arena.deinit();
    const allocator = arena.allocator();

    const spawn_config = std.Thread.SpawnConfig{ .allocator = allocator
        ↪ };

    const base_size = leaf_indices.len / num_jobs;
    var remainder = leaf_indices.len % num_jobs;

    var threads = allocator.alloc(std.Thread, num_jobs) catch @panic("
        ↪ error in allocation");

    // @bug workload is not balanced on the threads
    var cursor: usize = 0;
    for (threads) |*thread| {
        var true_size = base_size;
        if (remainder > 0) {
            true_size += 1;
            remainder -= 1;
        }
        const thread_data = Thread_Data{
            .ontology_tree = _ontology_tree,
            .leaf_indices = leaf_indices,
            .table_c2c = table_c2c,
            .start = cursor,
            .len = true_size,
```

```
            };
            cursor += true_size;
            thread.* = std.Thread.spawn(spawn_config, Thread_Data.job, .{
                ↪ thread_data}) catch @panic("cannot create thread");
        }

        for (threads) |thread| {
            thread.join();
        }
    }

    ontology = _ontology_tree;

    // returns
    py.Py_INCREF(py.Py_None);
    return py.Py_None;
}

// Utility function to convert python datasets of patients in plain arrays
fn parse_list_of_patients(codes: ?*py.PyObject, counts: ?*py.PyObject,
    ↪ allocator: std.mem.Allocator) ?[][][]u32 {
    if (py.PyList_Check(codes) == 0) {
        py.PyErr_SetString(generator_error, "Argument `codes` should be a
            ↪ list");
        return null;
    }
    if (py.PyList_Check(counts) == 0) {
        py.PyErr_SetString(generator_error, "Argument `counts` should be a
            ↪ list");
        return null;
    }

    const list_len = blk: {
        const _list_len = py.PyList_GET_SIZE(codes);
        if (_list_len != py.PyList_GET_SIZE(counts)) {
            py.PyErr_SetString(generator_error, "Argument `codes` and `
                ↪ counts` should have the same size");
            return null;
        }
        break :blk @as(usize, @intCast(_list_len));
    };

    const dataset = blk: {
        var dataset = allocator.alloc([][]u32, list_len) catch return null;
        for (0..list_len) |it| {
            const codes_obj = py.PyList_GetItem(codes, @intCast(it));
            const count_obj = py.PyList_GetItem(counts, @intCast(it));
            dataset[it] = parse_patient(codes_obj, count_obj, allocator)
                ↪ orelse {
                const msg = std.fmt.allocPrintZ(global_arena, "Error while
                    ↪ parsing dataset patient {}", .{it}) catch return null
                    ↪ ;
                py.PyErr_SetString(generator_error, msg);
                return null;
            };
        }
        break :blk dataset;
    };

    return dataset;
}
```

```
// Utility function to convert a patient from python object format to a
    ↪ plain array.
fn parse_patient(codes: ?*py.PyObject, counts: ?*py.PyObject, allocator: std
    ↪ .mem.Allocator) ?[][]u32 {
    const patient_id = blk: {
        const _patient_id = np.from_otf(codes, Np.Types.UINT, Np.Array_Flags
            ↪ .IN_ARRAY);
        break :blk @as(*Np.Array_Obj, @ptrCast(_patient_id orelse return
            ↪ null));
    };
    const patient_cc = blk: {
        const _patient_cc = np.from_otf(counts, Np.Types.UINT, Np.
            ↪ Array_Flags.IN_ARRAY);
        break :blk @as(*Np.Array_Obj, @ptrCast(_patient_cc orelse return
            ↪ null));
    };

    if (patient_id.nd != 1) return null;
    if (patient_cc.nd != 1) return null;

    const num_visits: usize = @intCast(patient_cc.dimensions[0]);
    var patient = allocator.alloc([]u32, num_visits) catch return null;

    const visit_lens: []u32 = @as([*]u32, @ptrCast(@alignCast(patient_cc.
        ↪ data)))[0..num_visits];
    const data: [*]u32 = @ptrCast(@alignCast(patient_id.data));

    var cursor: usize = 0;
    for (visit_lens, 0..) |c, it| {
        const len: usize = @intCast(c);
        patient[it] = data[cursor .. cursor + len];
        cursor += len;
    }

    const num_codes: usize = @intCast(patient_id.dimensions[0]);
    if (cursor != num_codes) return null;

    return patient;
}

// compute the distance of the two codes using the ontology
fn compute_c2c(id_1: u32, id_2: u32, _ontology_tree: []u32) f32 {
    if (id_1 == id_2) return 0;

    const res_1 = get_genealogy(id_1, _ontology_tree);
    const res_2 = get_genealogy(id_2, _ontology_tree);

    const genealogy_1 = res_1[0];
    const genealogy_2 = res_2[0];
    const root_1 = res_1[1];
    const root_2 = res_2[1];

    var cursor_1 = root_1;
    var cursor_2 = root_2;
    while (genealogy_1[cursor_1] == genealogy_2[cursor_2]) {
        if (cursor_1 == 0 or cursor_2 == 0) break;
        cursor_1 -= 1;
        cursor_2 -= 1;
    }
    cursor_1 = @min(cursor_1 + 1, root_1);
    cursor_2 = @min(cursor_2 + 1, root_2);
```

```
    const d_lr_doubled: f32 = @floatFromInt(2 * (root_1 - cursor_1));
    const dist = 1.0 - d_lr_doubled / (@as(f32, @floatFromInt(cursor_1 +
        ↪ cursor_2)) + d_lr_doubled);

    return dist;
}

// utility function to follow the genealogy on the ontology tree
fn get_genealogy(id: u32, _ontology_tree: []u32) struct { [
    ↪ genealogy_max_size]u32, usize } {
    var res = std.mem.zeroes([genealogy_max_size]u32);
    res[0] = id;
    var it: usize = 0;
    while (true) {
        const parent = _ontology_tree[res[it]];
        if (parent != res[it]) {
            it += 1;
            res[it] = parent;
        } else break;
    }
    return .{ res, it };
}

// compute the asymmetrical visit2visit distance
fn asymmetrical_v2v(v1: []u32, v2: []u32, _table_c2c: Table_c2c) f32 {
    var sum: f32 = 0;
    for (v1) |c1| {
        var best = std.math.floatMax(f32);
        for (v2) |c2| {
            const dist = _table_c2c.get(c1, c2);
            best = @min(best, dist);
        }
        sum += best;
    }
    return sum;
}

// compute the visit2visit distance
fn compute_v2v(v1: []u32, v2: []u32, _table_c2c: Table_c2c) f32 {
    const x = asymmetrical_v2v(v1, v2, _table_c2c);
    const y = asymmetrical_v2v(v2, v1, _table_c2c);
    return @max(x, y);
}

fn compute_p2p(p1: [][]u32, p2: [][]u32, _table_c2c: Table_c2c, allocator:
    ↪ std.mem.Allocator) f32 {
    var table = allocator.alloc(f32, p1.len * p2.len) catch @panic("error
        ↪ with allocation");

    const w = p1.len;

    for (0..p1.len) |it| {
        for (0..p2.len) |jt| {
            const cost = compute_v2v(p1[it], p2[jt], _table_c2c);
            var in_cost: f32 = std.math.floatMax(f32);
            var del_cost: f32 = std.math.floatMax(f32);
            var edit_cost: f32 = std.math.floatMax(f32);

            if (it > 0) {
                in_cost = table[jt * w + it - 1];
                if (jt > 0) {
                    del_cost = table[(jt - 1) * w + it];
```

```
                    edit_cost = table[(jt - 1) * w + it - 1];
                }
            } else {
                if (jt > 0) {
                    del_cost = table[(jt - 1) * w + it];
                } else {
                    edit_cost = 0;
                }
            }

            table[jt * w + it] = cost + @min(in_cost, del_cost, edit_cost);
        }
    }

    return table[table.len - 1];
}

/// result is a preallocated array for the result of the same size of
    ↪ dataset
fn compute_patients_distances(patient: [][]u32, dataset: [][][]u32,
    ↪ _table_c2c: Table_c2c, result: []f32) void {
    if (comptime num_jobs == 1) {
        var arena = std.heap.ArenaAllocator.init(std.heap.page_allocator);
        defer arena.deinit();
        const allocator = arena.allocator();

        for (dataset, 0..) |d_patient, it| {
            const dist = compute_p2p(patient, d_patient, _table_c2c,
                ↪ allocator);
            result[it] = dist;
            _ = arena.reset(.retain_capacity);
        }
    } else {
        const Thread_Data = struct {
            table_c2c: Table_c2c,
            dataset: [][][]u32,
            patient: [][]u32,
            result: []f32,
            current_index: *usize,

            const Self = @This();
            const batch_size = 16;

            pub fn job(self: Self) void {
                var arena = std.heap.ArenaAllocator.init(std.heap.
                    ↪ page_allocator);
                defer arena.deinit();
                const allocator = arena.allocator();

                while (true) {
                    const index: usize = @atomicRmw(usize, self.
                        ↪ current_index, .Add, Self.batch_size, .Monotonic)
                        ↪ ;
                    if (index >= self.dataset.len) break;

                    const index_limit = @min(self.dataset.len, index +
                        ↪ batch_size);
                    for (self.dataset[index..index_limit], index..) |
                        ↪ d_patient, it| {
                        const dist = compute_p2p(self.patient, d_patient,
                            ↪ self.table_c2c, allocator);
                        self.result[it] = dist;
```

```
                    _ = arena.reset(.retain_capacity);
                }
            }
        }
    };

    var arena = std.heap.ArenaAllocator.init(std.heap.page_allocator);
    defer arena.deinit();
    const allocator = arena.allocator();

    const spawn_config = std.Thread.SpawnConfig{ .allocator = allocator
        ↪ };
    var threads = allocator.alloc(std.Thread, num_jobs) catch @panic("
        ↪ error in allocation");

    var current_index: usize = 0;

    for (threads) |*thread| {
        const thread_data = Thread_Data{
            .table_c2c = _table_c2c,
            .dataset = dataset,
            .patient = patient,
            .result = result,
            .current_index = &current_index,
        };
        thread.* = std.Thread.spawn(spawn_config, Thread_Data.job, .{
            ↪ thread_data}) catch @panic("cannot create thread");
    }

    for (threads) |thread| {
        thread.join();
    }
}
}

/// perform temporal encodings on the dataset
fn ids_to_encoded(dataset: [][][]u32, encoded: [*]f32, max_label: u32,
    ↪ lambda: f32) void {
    const data_size = dataset.len;
    const index = Indexer(2).init(.{ data_size, max_label });

    @memset(encoded[0..index.size], 0);

    for (dataset, 0..) |patient, it| {
        var factor: f32 = 1.0;
        for (0..patient.len) |jt| {
            defer factor *= lambda;

            const visit_it = patient.len - jt - 1;
            for (patient[visit_it]) |c| {
                encoded[index.ix(.{ it, c })] += factor;
            }
        }
    }
}

// simple perturbation. Unused
fn independent_perturbation(dataset: [][][]u32, multiply_factor: usize,
    ↪ keep_prob: f32) *py.PyObject {
    var ids_list = py.PyList_New(@intCast(dataset.len * multiply_factor));
    var counts_list = py.PyList_New(@intCast(dataset.len * multiply_factor))
        ↪ ;
```

```
    var arena = std.heap.ArenaAllocator.init(std.heap.page_allocator);
    defer arena.deinit();
    const allocator = arena.allocator();

    var _rng = std.rand.DefaultPrng.init(42);
    var rng = _rng.random();

    for (dataset, 0..) |patient, it| {
        for (0..multiply_factor) |jt| {
            const new_patient = independent_perturb_patient(patient,
                ↪ keep_prob, allocator, rng);
            const list_id: isize = @intCast(multiply_factor * it + jt);
            py.PyList_SET_ITEM(ids_list, list_id, new_patient.ids);
            py.PyList_SET_ITEM(counts_list, list_id, new_patient.counts);
            _ = arena.reset(.retain_capacity);
        }
    }

    const return_tuple = blk: {
        var tuple = py.PyTuple_New(2);
        _ = py.PyTuple_SetItem(tuple, 0, ids_list);
        _ = py.PyTuple_SetItem(tuple, 1, counts_list);
        break :blk tuple;
    };
    return return_tuple;
}

// Ontological perturbation
fn ontological_perturbation(dataset: [][][]u32, multiply_factor: usize,
    ↪ keep_prob: f32, _ontology: []u32) *py.PyObject {
    var ids_list = py.PyList_New(@intCast(dataset.len * multiply_factor));
    var counts_list = py.PyList_New(@intCast(dataset.len * multiply_factor))
        ↪ ;

    var arena = std.heap.ArenaAllocator.init(std.heap.page_allocator);
    defer arena.deinit();
    const allocator = arena.allocator();

    var _rng = std.rand.DefaultPrng.init(42);
    var rng = _rng.random();

    for (dataset, 0..) |patient, it| {
        for (0..multiply_factor) |jt| {
            const new_patient = ontological_perturb_patient(patient,
                ↪ keep_prob, _ontology, allocator, rng);
            const list_id: isize = @intCast(multiply_factor * it + jt);
            py.PyList_SET_ITEM(ids_list, list_id, new_patient.ids);
            py.PyList_SET_ITEM(counts_list, list_id, new_patient.counts);
            _ = arena.reset(.retain_capacity);
        }
    }

    const return_tuple = blk: {
        var tuple = py.PyTuple_New(2);
        _ = py.PyTuple_SetItem(tuple, 0, ids_list);
        _ = py.PyTuple_SetItem(tuple, 1, counts_list);
        break :blk tuple;
    };
    return return_tuple;
}
```

```
const Numpy_Patient = struct {
    ids: *py.PyObject,
    counts: *py.PyObject,
};

fn independent_perturb_patient(patient: [][]u32, keep_prob: f32, allocator:
    ↪ std.mem.Allocator, rng: std.rand.Random) Numpy_Patient {
    const new_visits = allocator.alloc([]bool, patient.len) catch
        ↪ unreachable;
    var num_taken: usize = 0;
    for (patient, new_visits) |visit, *new_visit| {
        if (visit.len == 0) @panic("we cannot have empty visits in input");
        new_visit.* = allocator.alloc(bool, visit.len) catch unreachable;
        var num_visit_taken: usize = 0;
        for (new_visit.*) |*it| {
            const r = rng.float(f32);
            it.* = r < keep_prob;
            if (it.*) num_visit_taken += 1;
        }
        if (num_visit_taken == 0) {
            num_visit_taken = 1;
            new_visit.*[0] = true;
        }
        num_taken += num_visit_taken;
    }

    const ids = np.simple_new(1, @ptrCast(&num_taken), Np.Types.UINT) orelse
        ↪  unreachable;
    const counts = np.simple_new(1, @constCast(@ptrCast(&patient.len)), Np.
        ↪ Types.UINT) orelse unreachable;
    const ids_data: [*]u32 = @ptrCast(@alignCast(@as(*Np.Array_Obj, @ptrCast
        ↪ (ids)).data));
    const counts_data: [*]u32 = @ptrCast(@alignCast(@as(*Np.Array_Obj,
        ↪ @ptrCast(counts)).data));

    var cursor: usize = 0;
    var source_cursor: u32 = 0;
    for (new_visits, counts_data) |new_visit, *count| {
        var counter: u32 = 0;
        for (new_visit) |taken| {
            if (taken) {
                ids_data[cursor] = source_cursor;
                cursor  += 1;
                counter += 1;
            }
            source_cursor += 1;
        }
        count.* = counter;
    }

    return Numpy_Patient{
        .ids = ids,
        .counts = counts,
    };
}

fn is_in(array: []u32, val: u32) bool {
    for (array) |it| {
        if (it == val) return true;
    }
    return false;
}
```

```
fn ontological_perturb_patient(patient: [][]u32, keep_prob: f32, _ontology:
    ↪ []u32, allocator: std.mem.Allocator, rng: std.rand.Random)
    ↪ Numpy_Patient {
    const new_visits = allocator.alloc([]bool, patient.len) catch
        ↪ unreachable;
    var num_taken: usize = 0;

    var masked_parents = std.ArrayList(u32).initCapacity(allocator, patient.
        ↪ len) catch unreachable;

    for (patient, new_visits) |visit, *new_visit| {
        if (visit.len == 0) @panic("we cannot have empty visits in input");
        new_visit.* = allocator.alloc(bool, visit.len) catch unreachable;

        var num_visit_taken: usize = 0;

        for (visit, new_visit.*) |c, *it| {
            const p = _ontology[c];
            if (is_in(masked_parents.items, p)) {
                it.* = false;
            } else {
                const r = rng.float(f32);
                it.* = r < keep_prob;
                if (it.*) {
                    num_visit_taken += 1;
                }
                else masked_parents.append(p) catch unreachable;
            }
        }

        // This should *almost* never happen
        if (num_visit_taken == 0) {
            num_visit_taken = 1;
            new_visit.*[0] = true;
        }
        num_taken += num_visit_taken;
    }

    const ids = np.simple_new(1, @ptrCast(&num_taken), Np.Types.UINT) orelse
        ↪  unreachable;
    const counts = np.simple_new(1, @constCast(@ptrCast(&patient.len)), Np.
        ↪ Types.UINT) orelse unreachable;
    const ids_data: [*]u32 = @ptrCast(@alignCast(@as(*Np.Array_Obj, @ptrCast
        ↪ (ids)).data));
    const counts_data: [*]u32 = @ptrCast(@alignCast(@as(*Np.Array_Obj,
        ↪ @ptrCast(counts)).data));

    var cursor: usize = 0;
    var source_cursor: u32 = 0;
    for (new_visits, counts_data) |new_visit, *count| {
        var counter: u32 = 0;
        for (new_visit) |taken| {
            if (taken) {
                ids_data[cursor] = source_cursor;
                cursor  += 1;
                counter += 1;
            }
            source_cursor += 1;
        }
        count.* = counter;
    }
```

```
    return Numpy_Patient{
        .ids = ids,
        .counts = counts,
    };
}

// Required for python bindings
var generator_module = py.PyModuleDef{
    .m_base = .{
        .ob_base = .{ .ob_refcnt = 1, .ob_type = null },
        .m_init = null,
        .m_index = 0,
        .m_copy = null,
    },
    .m_name = "generator",
    .m_doc = "generator module",
    .m_size = -1,
    .m_methods = @constCast(&module_methods),
    .m_slots = null,
    .m_traverse = null,
    .m_clear = null,
    .m_free = null,
};

// Required for python bindings, entry point of the module
// Imports numpy too
pub export fn PyInit_generator() ?*py.PyObject {
    const module = py.PyModule_Create(@constCast(&generator_module));
    if (module == null) return null;

    generator_error = py.PyErr_NewException("generator.error", null, null);
    py.Py_XINCREF(generator_error);
    if (py.PyModule_AddObject(module, "error", generator_error) < 0) {
        py.Py_XDECREF(generator_error);
        {
            const tmp = generator_error;
            if (tmp != null) {
                generator_error = null;
                py.Py_DECREF(tmp);
            }
        }
        py.Py_DECREF(module);
        return null;
    }

    np = import_numpy() catch std.debug.panic("cannot import numpy", .{});

    _arena = std.heap.ArenaAllocator.init(std.heap.page_allocator);
    global_arena = _arena.allocator();

    return module;
}

fn import_numpy() !Np {
    const numpy = py.PyImport_ImportModule("numpy.core._multiarray_umath");
    if (numpy == null) return error.generic;
    const c_api = py.PyObject_GetAttrString(numpy, "_ARRAY_API");
    if (c_api == null) return error.generic;
    const PyArray_api = blk: {
        const t = py.PyCapsule_GetPointer(c_api, null);
        if (t == null) return error.generic;
```

```
        const ret: [*]?*void = @ptrCast(@alignCast(t));
        break :blk ret;
    };
    return Np.from_api(PyArray_api);
}

const std = @import("std");
const py = @import("python.zig");

const Np = @import("numpy_data.zig");
const Indexer = @import("tensor.zig").Indexer;
```

References

[1] Clinical classifications software (ccs) for icd-9-cm. `https://hcup-us.ahrq.gov/toolssoftware/ccs/ccs.jsp`. Accessed: 2024-01-11.

[2] Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein gan. (arXiv:1701.07875), December 2017. arXiv:1701.07875 [cs, stat].

[3] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. (arXiv:1409.0473), May 2016. arXiv:1409.0473 [cs, stat].

[4] James Betker, Gabriel Goh, Li Jing, Tim Brooks, Jianfeng Wang, Linjie Li, Long Ouyang, Juntang Zhuang, Joyce Lee, Yufei Guo, Wesam Manassra, Prafulla Dhariwal, Casey Chu, Yunxin Jiao, and Aditya Ramesh. Improving image generation with better captions.

[5] Francesco Bodria, Fosca Giannotti, Riccardo Guidotti, Francesca Naretto, Dino Pedreschi, and Salvatore Rinzivillo. Benchmarking and survey of explanation methods for black box models. (arXiv:2102.13076), February 2021. arXiv:2102.13076 [cs].

[6] L. Breiman, Jerome H. Friedman, Richard A. Olshen, and C. J. Stone. Classification and regression trees. *Biometrics*, 40:874, 1984.

[7] Edward Choi, Mohammad Taha Bahadori, Andy Schuetz, Walter F Stewart, and Jimeng Sun. Doctor ai: Predicting clinical events via recurrent neural networks. page 18.

[8] Edward Choi, Mohammad Taha Bahadori, Le Song, Walter F. Stewart, and Jimeng Sun. Gram: Graph-based attention model for healthcare representation learning. (arXiv:1611.07012), April 2017. arXiv:1611.07012 [cs, stat].

[9] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. (arXiv:1412.3555), December 2014. arXiv:1412.3555 [cs].

[10] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. (arXiv:1406.2661), June 2014. arXiv:1406.2661 [cs, stat].

[11] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001.

[12] Sarthak Jain and Byron C. Wallace. Attention is not explanation. (arXiv:1902.10186), May 2019. arXiv:1902.10186 [cs].

[13] Alistair E. W. Johnson, Lucas Bulgarelli, Lu Shen, Alvin Gayles, Ayad Shammout, Steven Horng, Tom J. Pollard, Sicheng Hao, Benjamin Moody, Brian Gow, Li-wei H. Lehman, Leo A. Celi, and Roger G. Mark. Mimic-iv, a freely accessible electronic health record dataset. *Scientific Data*, 10(1):1, January 2023.

[14] Diederik P. Kingma and Max Welling. An introduction to variational autoencoders. *Foundations and Trends® in Machine Learning*, 12(4):307–392, 2019. arXiv:1906.02691 [cs, stat].

[15] Zachary C. Lipton. The mythos of model interpretability. *Commun. ACM*, 61(10):36–43, 2018.

[16] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization, 2019.

[17] Fenglong Ma, Radha Chitta, Jing Zhou, Quanzeng You, Tong Sun, and Jing Gao. Dipole: Diagnosis prediction in healthcare via attention-based bidirectional recurrent neural networks. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, page 1903–1911, August 2017. arXiv:1706.05764 [cs].

[18] Fenglong Ma, Quanzeng You, Houping Xiao, Radha Chitta, Jing Zhou, and Jing Gao. Kame: Knowledge-based attention model for diagnosis prediction in healthcare. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*, page 743–752, Torino Italy, October 2018. ACM.

[19] Cecilia Panigutti, Alan Perotti, Andrè Panisson, Paolo Bajardi, and Dino Pedreschi. Fairlens: Auditing black-box clinical decision support systems. (arXiv:2011.04049), November 2020. arXiv:2011.04049 [cs].

[20] Cecilia Panigutti, Alan Perotti, and Dino Pedreschi. Doctor xai: an ontology-based approach to black-box sequential data classification explanations. In *Proceedings of the 2020 Conference on Fairness, Accountability, and Transparency*, page 629–639, Barcelona Spain, January 2020. ACM.

[21] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks.

[22] Xueping Peng, Guodong Long, Tao Shen, Sen Wang, and Jing Jiang. Sequential diagnosis prediction with transformer and ontological representation. (arXiv:2109.03069), September 2021. arXiv:2109.03069 [cs].

[23] Jorge Pérez, Javier Marinković, and Pablo Barceló. On the turing completeness of modern neural network architectures. (arXiv:1901.03429), January 2019. arXiv:1901.03429 [cs, stat].

[24] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models. (arXiv:2302.13971), February 2023. arXiv:2302.13971 [cs].

[25] Hugo Touvron, Louis Martin, and Kevin Stone. Llama 2: Open foundation and fine-tuned chat models.

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

[27] Chulhee Yun, Srinadh Bhojanapalli, Ankit Singh Rawat, Sashank J. Reddi, and Sanjiv Kumar. Are transformers universal approximators of sequence-to-sequence functions? (arXiv:1912.10077), February 2020. arXiv:1912.10077 [cs, stat].

[28] Biao Zhang and Rico Sennrich. Root mean square layer normalization. (arXiv:1910.07467), October 2019. arXiv:1910.07467 [cs, stat].

[29] Min-Ling Zhang and Zhi-Hua Zhou. A review on multi-label learning algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 26(8):1819–1837, 2014.