UNIVERSITÄT
LEIPZIG

Faculty of Mathematics and Computer Science
Department of Computer Science
Database Group

Explorative Graph-Level Embeddings for Temporal Property Graphs

Bachelor Thesis

submitted by:
Linus Andreas Schneider

Matriculation number:
3705638

Supervision:
University of Leipzig: Dr. Christopher Rost
Neo4j: Kevin Gomez

©2024

This work including its parts is **protected by copyright**. Any use outside the narrow limits of copyright law without the consent of the author is prohibited and punishable. This applies in particular to duplications, translations, microfilming and storage and processing in electronic systems.

Abstract

This thesis explores methods for generating graph level embeddings of temporal property graphs containing electronic health record data by extending Neo4j's Graph Data Science Library implementation of FastRP node embeddings. Using a synthetic dataset generated with Synthea, various approaches for preprocessing string and temporal properties, generating node embeddings, and aggregating them into graph level representations were implemented and evaluated. The evaluation assessed effectiveness in capturing graph characteristics, scalability, and clustering performance. Results showed that using a sentence transformer model effectively captured semantic differences between graphs, while a pre-trained time2vec layer successfully encoded temporal aspects, particularly relative time differences between medical encounters. Mean aggregation proved viable for transforming node embeddings into graph level representations, with weighted aggregation enabling domain-knowledge-based tuning. While embeddings based on string properties showed strong clustering performance, incorporating temporal features created more uniform distributions in the embedding space, suggesting that temporal patterns in patient journeys, while meaningfully captured, may be too variable for effective density-based clustering. An interactive Jupyter notebook was developed to facilitate the exploration of embedding parameters and their effects on the representation of patient journeys in the embedding space.

Linus Andreas Schneider
3705638

Contents

List of Figures

List of Tables

Listings

List of Abbreviations

APOC	Awesome Procedures On Cypher
DBCV	Density-Based Clustering Validation
EHR	Electronic Health Record
GDBMS	Graph Database Mangement Systems
GDS	Neo4j Graph Data Science Library
HDBSCAN	Hierarchical Density-Based Spatial Clustering of Applications with Noise
NLP	Natural Language Processing
t-SNE	T-Distibuted Stochastic Neighbor Embedding
UMAP	Uniform Manifold Approximation and Projection

1. Introduction

1.1. Motivation

The adoption of *electronic health record systems* has been growing worldwide [1], leading to an increased availability of **electronic health record** (EHR) data. Since there are inherent connections present in such data, organizing it in a graph format allows for an intuitive and descriptive representation of it, explicitly encoding these connections as relationships between medical entities (patients, providers, diagnoses, drugs, etc.). Additionally EHR data has a strong temporal aspect, which can also be encoded as part of the graph.

For storing, managing, and extracting insights from large amounts of graph data, **graph database management systems** (GDBMS) offer many advantages. The GDBMS **Neo4j** being the most popular [2] choice and offering many avenues of generating insights from graph data. However, it doesn't yet offer a way of creating embeddings as graph level representations of entire (sub)graphs, which could be used to embed graphs representing a patients EHR, called **patient journeys**, for similarity search and potential applications in downstream clinical tasks. This leaves space for the exploration of ways of doing so involving the use of the Neo4j ecosystem in this thesis.

1.2. Aims and Objectives

This thesis mainly concerns itself with the exploration of different ways of generating graph level embeddings of patient journeys through the use and extension of functionalities implemented within the Neo4j ecosystem.

It aims to do so both through providing an interactive way of exploring the effects of different choices of preprocessing methods and embedding parameters on the results, as well as by deploying a ground truth for evaluating how effective different methods are in capturing aspects of the EHR data, with special consideration given to its temporal aspects.

The data, which forms the basis of the embeddings, will be generated synthetically through methods that accurately model real patient data, containing information about various medical entities. This will then be transformed into a graph representation inside Neo4j. This representation will be the basis for the exploration of different approaches of creating graph level embeddings patient journeys, making use of an interactive implementation to provide an intuitive understanding, as well as a more rigorous evaluation.

The embeddings will be evaluated with regard to their effectiveness and the scalability of the methods used to generate them in order to provide insight into their applicability, avenues for their improvement, and further investigation in future work.

1.3. Thesis Outline

The thesis starts with a comprehensive explanation of its background in Chapter 2, covering all the concepts needed for the understanding and interpretation of its content and underlying implementation. Then in Chapter 3, related work in the fields of graph level embeddings and EHR embeddings is presented, to provide an overview of the scientific context this work is placed in. Afterwards the datasets underlying model, generation, and features are described and explored in Chapter 4, followed by a description of the design and implementation of the interactive way of generating the graph level embeddings in Chapter 5. Lastly, the efficacy of different methods is evaluated in Chapter 6, where its results are discussed, while Chapter 7 provides a summary and interpretation of the thesis' content and findings, with Section 7.2 exposing avenues for further research, based on the findings and limitations of this work.

2. Background

This thesis is based on an understanding of graph theory, embeddings, techniques of clustering and visualizing them, and the Neo4j ecosystem, the implementation of FastRP in particular. This chapter will provide an overview of these topics and their relevance to the thesis, as well as an explanation of the hyperparameters for the algorithms used.

2.1. Graphs

2.1.1. Undirected Simple Graphs

The simplest form of a graph is an **undirected simple graph**. It consists of a finite set $V \neq \emptyset$ of nodes, also called vertices, and a set $E \subseteq \{\{u, v\} \mid u, v \in V, u \neq v\}$ of edges, also called relationships, and can be denoted as $G = (V, E)$.

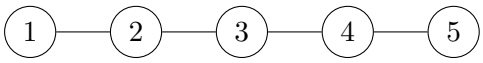


Figure 2.1.: Example of an Undirected Simple Graph.

A *path* in a graph is a sequence of nodes connected by edges, where each node is connected to the next by an edge. The *degree* $d(v)$ of a node v is the number of edges connected to it, describing the number of *neighbors* a node has. It can be used to define the **degree matrix** of a graph, which is a diagonal matrix containing the degrees of the nodes on its diagonal and zeros elsewhere. The degree matrix of the graph in Figure 2.1 would be:

$$D = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

2.1.2. Directed Multigraphs

A more expressive version of graphs are **directed multigraphs**. They can be denoted as $G = (V, E, s, t)$ and differ from undirected subgraphs in that E is a *multi*set of edges, meaning that multiple edges between two nodes are allowed. Additionally it contains the maps $s : E \to V$ and $t : E \to V$ assigning to each edge a source and target vertex, which makes them directed.

Directed graphs can also be represented as an *adjacency matrix*, which is a square $|V| \times |V|$ matrix S, where S_{ij} is 1 if there is an edge from node i to node j and 0 otherwise. The adjacency matrix of the graph in Figure 2.2 would be:

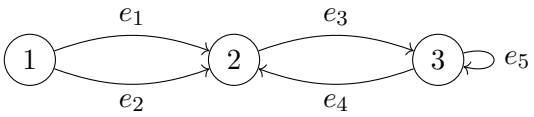


Figure 2.2.: Example of a Directed Multigraph.

$$S = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}$$

Adjacency matrices can also be used to represent undirected graphs. In that case the matrix is symmetric, meaning that $S_{ij} = S_{ji}$ for all $i, j \in V$.

2.1.3. Temporal Property Graphs

In order to be able to more accurately model real-world scenarios the directed multigraph can be extended by:

- Adding a set $B = \{b_1, b_2, \ldots, b_n\}$ of type labels
- A map $\beta : (V \cup E) \to B$ assigning a single label/type to every vertex or edge
- A set of property keys $K = \{k_1, k_2, \ldots, k_n\}$ and a set of data values $A = \{a_1, a_2, \ldots, a_n\}$
- A partial function $\kappa : (V \cup E) \times K \to A$ assigning data values to vertex/edge and property key combinations
- Changing each vertex to be a tuple (v, τ^{val}), where $\tau^{val} = [\omega_{start}, \omega_{end})$ is the valid time of the vertex, consisting of a start time and optional end time, or being empty.

The notion of using the property keys K, values A, and corresponding mapping κ was proposed by Angles et al. as the **property graph** model [3]. and is now widely used in graph databases like Neo4j. The extension of the property graph model to include the valid time of a node was proposed by Rost et al. as the **temporal property graph** model [4], which also includes notions of *transaction time*, *graph collections*, and *logical graphs*, which are not relevant for this work and therefore have been omitted.

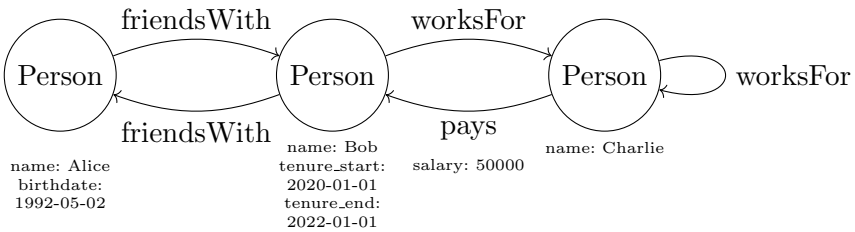


Figure 2.3.: Example of a Social Network as a Temporal Property Graph.

2.2. Embeddings

An embedding function emb : $\mathcal{X} \to \mathbb{R}^d$ maps objects from a set $\mathcal{X}$ to d-dimensional vectors. For an object $O \in \mathcal{X}$, its embedding is the vector $\text{emb}(O) \in \mathbb{R}^d$. Much of the usefulness of embeddings comes from the fact that they can be used to compare objects in the embedding space using mathematical similarity measures, which is especially useful when applied to complex objects which couldn't otherwise be easily compared, like temporal property graphs.

2.2.1. Similarity Measure

Generally, similarity measures are some form of (distance) metric between two embedding vectors, which can be used to compare the objects they represent.

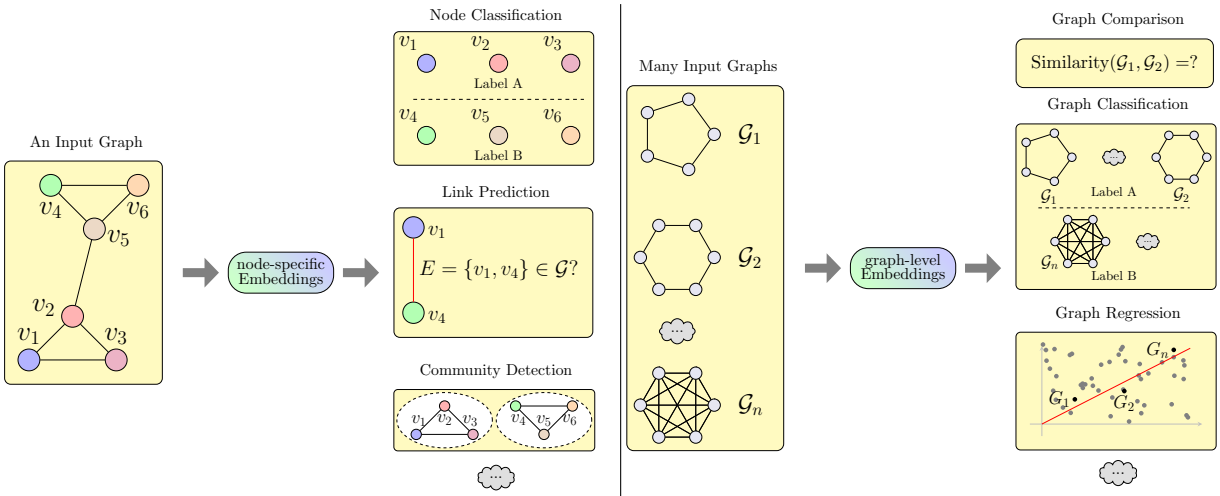
The similarity measure chosen for this work is the cosine similarity, defined as:

$$\text{Similarity}(v_1, v_2) = \cos(\theta) = \frac{v_1 \cdot v_2}{|v_1||v_2|} \tag{2.1}$$

where $v_1 \cdot v_2$ denotes the dot product and $|v|$ the Euclidean norm of vector v. It encodes the angle between two vectors, which is 1 if they are identical, 0 if they are orthogonal, and -1 if they are opposite to each other and is invariant to the magnitude of the vectors. It was chosen because of it's simplicity, because it doesn't require explicit normalization of the embedding vectors, and has been used in other works [5] [6] exploring graph level embeddings of temporal graphs.

2.2.2. Graph Embeddings

Any embedding that takes a graph as input is called a graph embedding. However, there are two types of graph embeddings, depending on the level of the graph they are applied to, the difference between the two being illustrated in Figure 2.4 below.



(a) Applications of node-specific Graph Embeddings.

(b) Applications of graph-level Graph Embeddings.

Figure 2.4.: Application of Node Level and Graph Level Graph Embeddings, adapted from [7].

Node level embeddings capture information about individual nodes, such as their properties, connections, and roles within the graph. However, they often miss the global structure of the graph, while **graph level embeddings** capture information about the entire graph, including its structure and node relationships. These embeddings often aggregate node level embeddings through a process known as *graph pooling* [7].

2.2.3. Text Embeddings

Commonly, properties of temporal property graphs are textual, like the name of a person or the type of a relationship. In order to use these properties in the graph embeddings, they need to be converted to numerical vectors, which is done using a form of natural language processing (NLP) known as *text embedding*. Through text embedding, a string of text, which can be a single word, a phrase, or a whole paragraph, is converted to a vector representation, which aims to capture the meaning of the text in a high-dimensional space.

In this work a *transformer model* [8] was used to generate the text embeddings. More specifically, a specialized version of the bidirectional encoder-only transformer model *BERT* [9] known as **sentence transformer** [10] was used, which was trained on a large corpus of text data to generate sentence embeddings that allow for an effective comparison of sentences using cosine similarity.

The sentence transformer processes text through multiple neural network layers that analyze the relationships between words within their context. These relationships are captured in the model's *attention weights*, which are then used to generate sentence embeddings by aggregating word embeddings based on their contextual importance. The resulting fixed-length vector representations encode semantic meaning in a way that similar phrases result in similar vectors, making them suitable for comparison operations. The exact functioning of the transformer architecture exceeds the scope of this thesis, as the focus lies on the application of the resulting embeddings rather than their generation process.

2.2.4. Temporal Embeddings

As this work investigates *temporal* property graphs, the embeddings generated from the graphs need to be able to capture the temporal aspects of the data. Time has both *linear*, as well as a *periodic* aspect, which are both relevant. Age increases linearly throughout a persons life but patterns like the day of the week, flu-season, or the time of day are periodic and can be just as relevant for the analysis of the data. **Time2vec** [11] introduced a simple yet effective way of encoding both of these aspects in a single neural network layer.

$$\mathbf{t2v}(\tau)[i] = \begin{cases} \omega_i \tau + \varphi_i, & \text{if } i = 0 \\ \mathcal{F}(\omega_i \tau + \varphi_i), & \text{if } 1 \leq i \leq k \end{cases} \tag{2.2}$$

It does this by taking a periodic input variable $\tau \in \mathbb{R}^k$, multiplying it with a set of learnable weights ω_i and adding a set of learnable biases φ_i for each dimension i of the output vector. For the first dimension, the output is linear, while for the other dimensions it is passed through a periodic

activation function $\mathcal{F}$, which is usually a sine or cosine function, enabling it to learn both linear and periodic patterns in the data. The layer can either be used as a standalone layer or as part of a larger neural network.

2.3. Dimensionality Reduction

While typically larger embeddings are able to capture more information about the objects they represent, they can also be harder to work with, as they don't have an intuitive visual representation. This is solved through the use of dimensionality reduction techniques, which preserve the relationships between the data, while reducing it to dimensions that can be visualized.

2.3.1. Uniform Manifold Approximation and Projection

One such method is *Uniform Manifold Approximation and Projection*, commonly referred to as **UMAP** [12]. Compared to other popular dimensionality reduction techniques, like *T-Distributed Stochastic Neighbor Embedding* (t-SNE) [13], it is much much more efficient and scales much better. It is also able to preserve the global structure of the reduced embeddings better than t-SNE.

It constructs a neighborhood graph for the high dimensional data by first drawing a variable sized radius around every data point, based on the distance of the k-th nearest neighbor of that point, where k a hyperparameter (called `n_neighbours` in the scikit-learn implementation used in this work) that can be used to set the focus on more local or more global structures. Then, between data points with overlapping radii an edge is added, which has a weight symbolizing the connection probability based on the distance between the connected points.

In the second part of the algorithm the high-dimensional neighborhood graph is projected into a lower dimensional space, while mostly preserving the distances between connected points. To do this for every edge the cross entropy between its weight in the high-dimensional space $w_h(e)$ and its weight in the low-dimensional space $w_l(e)$

$$\sum_{e \in E} w_h(e) \log\left(\frac{w_h(e)}{w_l(e)}\right) + (1 - w_h(e)) \log\left(\frac{1 - w_h(e)}{1 - w_l(e)}\right) \tag{2.3}$$

is minimized. In the equation the first summand $w_h(e) \log\left(\frac{w_h(e)}{w_l(e)}\right)$ of the term essentially functions as an attractive force between the connected points, as it's lowest, when the weight is lowest in the low dimensional space, while the second summand $(1 - w_h(e)) \log\left(\frac{1-w_h(e)}{1-w_l(e)}\right)$ functions as a repulsive force, as it's lowest when the weight is largest in the low dimensional space. The value for $w_l(e)$ can be bounded by a hyperparameter `min_dist`, which if low causes points to clump closer together, reflecting harder, more local features, while if high causes points to spread out more, reflecting softer, more global features.

2.4. Clustering

Clustering describes the task of grouping objects together based on their similarity, with the goal of maximizing the similarity within clusters and minimizing the similarity between them. There are numerous clustering algorithms, which can be grouped into different categories based on their underlying approaches. *Centroid-based* methods cluster points around iteratively updated centers but require pre-specification of cluster numbers. *Connectivity-based* methods build hierarchical relationships between points but are computationally intensive for large datasets. *Density-based* methods identify clusters as dense regions in the data space and can handle irregular shapes, though they require careful parameter tuning.

2.4.1. Hierarchical Density-Based Spatial Clustering of Applications with Noise

For the analysis in this work the *Hierarchical Density-Based Spatial Clustering of Applications with Noise* algorithm, **HDBSCAN** [14] for short, was selected. It combines density-based and hierarchical approaches, providing the benefits of density-based clustering while minimizing the need for parameter tuning. The algorithm first calculates the mutual reachability distance between points a and b using their k-th nearest neighbor distances (core_k):

$$d_{\text{mreach-k}}(a, b) = \max\{\text{core}_k(a), \text{core}_k(b), d(a, b)\} \tag{2.4}$$

where $d(a, b)$ is the original metric distance between the points.

Using these distances, it builds and processes a hierarchical tree of clusters, merging and splitting them based on their size and stability. Three main hyperparameters control this process: `min_cluster_size` sets the minimum size for valid clusters, `min_samples` determines how conservative the algorithm is in declaring clusters versus noise points, and `cluster_selection_method` chooses between selecting the most stable clusters (`eom`) or the leaf clusters (`leaf`) from the hierarchy.

2.4.2. Performance Evaluation

Clustering performance can be validated using either *external* or *internal* measures. External measures compare the clustering to known class structures, but require pre-existing knowledge of the correct grouping, while internal measures evaluate cluster cohesion and separation without reference data. However, internal measures often struggle with non-globular clusters and noisy data.

Density-Based Clustering Validation, referred to as **DBCV** [15], was developed specifically for validating density-based clustering algorithms, where no ground truth is available and the clusters are not necessarily globular.

It defines the DBCV index of a clustering C

$$\text{DBCV}(C) = \sum_{i=1}^{i=l} \frac{|C_i|}{|O|} V_C\left(C_i\right) \tag{2.5}$$

as a weighted average of the validity index V_C of all clusters $C_i \in C$, where $|C_i|$ is the clusters size and $|O|$ is the total number of objects including those classified as noise, meaning a lower noise-ratio leads to higher values.

The validity index V_C of a single cluster C_i is

$$V_C(C_i) = \frac{\min_{1 \leq j \leq 1, j \neq i} (\mathrm{DSPC}(C_i, C_j)) - \mathrm{DSC}(C_i)}{\max(\min_{1 \leq j \leq l, j \neq i} (\mathrm{DSPC}(C_i, C_j)), \mathrm{DSC}(C_i))} \tag{2.6}$$

where $\mathrm{DSC}(\mathrm{C_i})$ denotes the *Density Sparseness* of C_i, encoding the lowest density within the cluster, and $\mathrm{DSPC}(\mathrm{C_i}, \mathrm{C_j})$ the *Density Separation* between it and another cluster C_j, meaning the highest density between the two clusters. This means V_C is highest for a cluster C_i when it is well separated from other clusters and has a high density within itself.

The DBCV index of a clustering is then a value between -1 and 1, where greater values indicate denser, better separated clusters overall. While it does provide a robust measure of cluster quality, the DBCV index does *not* provide any information about the usefulness or expressiveness of the clusters for a specific task or domain. Without expert knowledge of the domain and within the constraints of unsupervised learning, however, it seems to be the best measure available for evaluating the quality of density-based clustering in a quantifiable way.

2.5. Neo4j

Neo4j [16] is the GDBMS used to store and manage the data used in this work. It offers performance and flexibility in handling complex graph data and extracting insights from it [17].

2.5.1. Interacting with Neo4j

Cypher [18] is the query language used to interact with Neo4j databases. It is a declarative, pattern-matching language, whose queries are based on ascii-art patterns that describe the structure of the graph in an intuitive way.

```
1 MATCH (n:Node)−[r:REL]−>(m:Node)
2 WHERE n.property = ”value”
3 RETURN n, r, m
```

Listing 2.1: Example of a Simple Cypher Query

Additionally to basic querying and aggregation functions it offers a variety of options for creating more complex requests, like chaining queries, subqueries, conditional statements, unwinding lists, and using parameters, as well as implementing basic functions, for example getting the id of a node, or the time between two DateTime values. The language is very expressive and well documented, which is why specifics will not be further elaborated on here.

Besides the more basic functions that can be accessed directly through Cypher, Neo4j also offers the **APOC** [19] library, *Awesome Procedures On Cypher*, which provides a wide range of optimized additional functions and procedures to interact with the database and the *Neo4j Graph Data Science Library* [20], henceworth referred to as **GDS**, which provides efficiently implemented and highly optimized versions of various graph algorithms.

A particularly useful subset of said algorithms are various node embedding algorithms. In order to utilize them, a given (sub)graph needs to be projected into an in-memory version of the graph, which can then be used for generating node embeddings of it. Additionally, a python client is provided for GDS, which can be used to run Cypher queries and returns the results as a pandas DataFrame, which can then be used to further process the data from within the python environment.

2.5.2. FastRP

The node embedding algorithm provided by GDS with the best performance and scalability [21] is the *Fast Random Projection* algorithm, **FastRP** [22].

It gets its name from the fact that it's multiple orders of magnitude faster than other node embeddings algorithms, like DeepWalk [23] or Node2Vec [24]. This is achieved through leveraging the optimization-free dimensionality reduction technique of *Very Sparse Random Projection* [25], which is able to project high-dimensional data into a lower-dimensional space while preserving the pairwise distances between the data points.

As discussed in Section 2.1, a graph can be represented as an $n \times m$ adjacency matrix S and a diagonal $n \times m$ degree matrix D can be computed for it. By multiplying the *inverted* degree Matrix D with the adjacency matrix S the *transition matrix*

$$A = D^{-1}S \tag{2.7}$$

is formed, where each entry A_{ij} of A contains the probability of reaching node j from node i in one random step. Building upon this a *k-step transition matrix* A^k is computed, for which every ij-th entry contains the probability of reaching node j from node i through a k-step random walk, thereby encoding the graphs neighborhood structures.

However, calculating A^k is too costly of an operation. This is circumvented by FastRP by the construction of a *reduced $m \times d$ k-step transition matrix* A_r^k where $d < n$, which has the same pairwise distances between data points as A^k but can be calculated much more efficiently through the use of the associative property of matrix multiplication and a *random projection matrix* R whose values are randomly sampled from a zero-mean distribution by setting

$$R_{ij} = \begin{cases} \sqrt{s} & \text{with probability } \frac{1}{2s} \\ 0 & \text{with probability } 1 - \frac{1}{s} \\ -\sqrt{s} & \text{with probability } \frac{1}{2s} \end{cases} \tag{2.8}$$

where $s = n$. Thereby, the reduced transition matrix can be calculated as

$$\underbrace{(A \cdot A \cdot \ldots \cdot A)}_{k} \cdot R = A_r^k = \underbrace{(A \cdot \ldots (A \cdot (A}_{k} \cdot R))) \tag{2.9}$$

The dimension d of R and therefore the dimension of the generated node embeddings, which are the rows of the resulting matrix, can be set using the hyperparameter `embeddingDimension`.

Additionally, normalization is applied through multiplication with a $n \times n$ *normalization matrix*

$$L = \operatorname{diag}\left(\left(\frac{d_1}{2m}\right)^{\beta}, \ldots, \left(\frac{d_n}{2m}\right)^{\beta}\right) \tag{2.10}$$

where d_j is the out-degree of a node j and β, which can be set through the hyperparameter `normalizationStrength` and controls how much each nodes degree should influence the final embeddings. The higher it is the more the final embeddings will be influenced by more connected nodes, the lower it is, the more equal the influence of all nodes will be.

This results in a *reduced, normalized k-step transition matrix*

$$\widetilde{A}^k = (A \cdot \ldots (A \cdot (A \cdot L \cdot R))) \tag{2.11}$$

Finally the embeddings are aggregated into a final embedding matrix N through the weighted combination of different powers of the transition matrix:

$$N = (\alpha_1 \widetilde{A}^1 + \alpha_2 \widetilde{A}^2 + \ldots + \alpha_k \widetilde{A}^k) \tag{2.12}$$

where the weights α_i can be set through the hyperparameter `iterationWeights` and control how much influence the k-th neighbors of a given node have on the final embedding. The length of the array determines the number of iterations k that are used to generate the final embeddings.

The GDS implementation extends the functionality of the original FastRP algorithm described above by allowing R to be, either partly or fully, deterministically generated from a set of properties of the nodes, specified through the hyperparameter `featureProperties`. The ratio of how many columns of R are based on the node properties and how many are randomly sampled can be set through the hyperparameter `propertyRatio`. The properties need to be either scalars or arrays of scalars. For every scalar x (even if it's inside an array) a very sparse random vector v_i of dimension propertyRatio $\cdot$ embeddingDimension is generated. These vectors are then linearly combined to form the node specific row R_i by summing over all n scalars x_i and v_i:

$$R_i = \sum_{i=1}^{n} x_i \cdot v_i \tag{2.13}$$

This process is deterministic because a seed for all randomness in the algorithm, including the sampling of the very sparse random vectors, can be set through the hyperparameter `seed`.

Since this causes R to encode information about the nodes instead of just random information, the embedding aggregation was extended to include it inside the final embedding matrix N as well, by adding it to the weighted combination of the different powers of the transition matrix:

$$N = (\alpha_0 R \cdot L + \alpha_1 \widetilde{A}^1 + \alpha_2 \widetilde{A}^2 + \ldots + \alpha_k \widetilde{A}^k) \tag{2.14}$$

where the weight α_0 can be set through the hyperparameter `nodeSelfInfluence`.

Additionally, the algorithm supports weighted graphs through the `relationshipWeightProperty` parameter. When specified, edge weights influence how strongly connected nodes contribute to each other's embeddings throughout the iterative multiplication steps. The algorithm's normalization ensures numerical stability despite potentially large differences in edge weights.

3. Related Work

This thesis lies at the intersection of two distinct areas of research: *graph level embeddings* and *EHR embeddings*. While graph level embeddings are often more theoretical in nature and concern themselves mostly with the topological properties of graphs, EHR embeddings focus more on the domain-specific challenges of electronic health record data.

Graph level embeddings represent a more established field with a longer history, drawing from graph theory and utilizing classical machine learning techniques. In contrast, EHR embeddings are a more recent development that has seen a surge of interest in the past few years, driven by advances in deep learning techniques [9] and the availability of large-scale healthcare datasets [1].

Despite their differences, both areas share a common goal: To learn meaningful representations of complex data structures, such as graphs or sequences, in a way that preserves the inherent properties of the data. These representations enable more effective downstream tasks, such as clustering or classification.

3.1. Graph Level Embeddings

Yang et al. [7] provide an excellent overview of the history and current state of graph level learning. Notable examples of graph-level learning techniques include:

- **Graph2vec [26]:** Samples rooted subgraphs around every node of the graph. Then uses skipgram with negative sampling akin to *doc2vec* [27] on those samples.
- **UGraphEmbed [28]:** Uses a *graph isomorphism network* [29] to create node embeddings and uses *multi-scale node attention* to aggregate them in a way that the embeddings of two gaphs preserve their proximity based on the *graph edit distance*.
- **NetLSD [30]:** Uses the *heat trace* of the *laplacian eigenspectrum* of the adjacency matrix representing the graph/the heat kernel of the graph.

Most graph level embedding techniques are limited to capturing the static topology of a graph. While there are some advanced methods like FEATHER-G [31] that are able to capture node properties, they are still limited to static graphs.

3.1.1. Temporal Graph Level Embedding

There have been some graph level embedding techniques developed to explicitly capture the temporal evolution of a graph. For example the following techniques:

- **Temporal backtracking random walk [6]**: Uses a *multilayer graph*, where every time step in the graphs evolution is represented by one layer, and *random walks* with the ability to change layers, which are then aggregated similarly to the method of *graph2vec* [26], optimizing the probability of seeing a node within its context, which is defined as a sliding window over its random walk.
- **GraphERT [32]:** Uses multiple *random walks* and a *biderectional encoder-only transformer* architecture trained simultaneously on predicting masked nodes in those random walks and the time step of each random walk, in order to learn a representation that captures both the topology and the temporal evolution of the graph through the attention weights.

These do not, however, offer the ability to incorporate node attributes into the embeddings. Additionally, not every graph with temporal features can be meaningfully divided into time steps.

3.2. Electronic Health Record Embeddings

Both the temporal nature of EHR data and its domain-specific representation are of particular importance to learning meaningful embeddings. The work of Li et al. [33] introduced the **BEHRT** model, which models EHR data akin to natural language data by treating diagnoses codes as words, patient visits as sentences, and patients EHR as documents. Regarding visit data they limited themselves to diagnoses codes, which they represented according to the ICD-10 and included temporal features by encoding the sequence of visits using positional embeddings and the relative time between visits by embedding the patients age at the time of the visit. The BEHRT model was trained to predict masked diagnoses codes in the visit sequence. Using the aggregated attention weights of the trained model as a EHR level embedding, it was able to outperform other state-of-the-art models on a variety of downstream classification/prediction tasks in the clinical domain.

There have been numerous works improving upon the BEHRT model. For example by applying recent findings from the field of NLP to the clinical domain, incorporating more learning tasks [34], and by utilizing more effective ways of encoding temporal information [35]. Notably, one work used a *graph transformer model* to learn relationship-aware embeddings of the visits in the EHR data [36], showing that the incorporation of graph representations of EHR data can be beneficial for learning meaningful embeddings.

4. Dataset

The dataset is the foundation of the generated embeddings and therefore plays a crucial role in the evaluation of the proposed methods. This chapter provides an overview of the dataset used in this thesis, including how it was created, its structure, and the information it contains about the application domain.

4.1. Patient Journey Model

A *Patient Journey* describes a sequence of medical encounters, associated events and data for a single patient over time. In this work when referring to a patient journey, it is meant as a temporal property graph, where the nodes represent patients that have encounters, which in turn are connected to other nodes representing the associated medical entities of medical observations, providers, diagnosed conditions, prescribed drugs, and medical procedures. Each patient's journey through the medical system is a path through the encounter nodes and their associated entities, as shown in simplified form in Figure 4.1.

Besides the explicit ordering of the encounter nodes in the graph through the `NEXT` relationships, the temporal aspect of the patient journey is encoded in the start and end times of the encounters, which encode their absolute temporal position, as well implicitly encoding the time elapsed between them and therefore their relative temporal position in the journey.

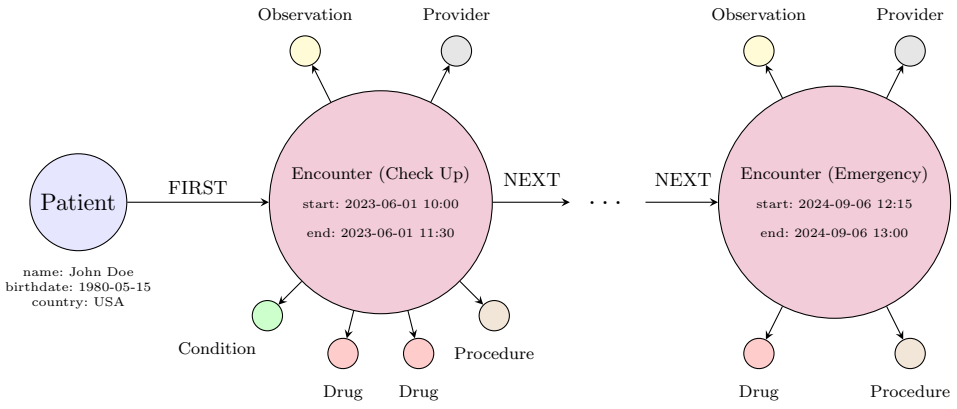


Figure 4.1.: Simplified Representation of a Patient Journey as a Temporal Property Graph.

Representing a patient's EHR data as a temporal property graph has the advantage of capturing the complex relationships between the different entities through encoding them directly within the topology of the graph.

In the example above, many properties and relationships of the patient journey model have been omitted, for the sake of clarity. The following is a full list of the (optional) properties of the entities contained in the patient journey and their respective data types:

Patient :
- id : String
- birthdate : DateTime
- birthplace : String
- first : String
- last : String
- gender : String
- race : String
- ethnicity : String
- city : String
- state : String
- county : String
- healthcare_expenses : Float
- address : String
- (deathdate : DateTime)

Encounter :
- id : String
- description : String
- code : String
- class : String
- start : DateTime
- (end : DateTime)
- basecost : Float
- isEnd : Boolean

Provider :
- id : String
- name : String
- address : String
- specialty : String

Condition :
- code : String
- description : String
- start : DateTime
- (end : DateTime)

Drug :
- code : String
- description : String
- start : DateTime
- (end : DateTime)
- basecost : Float

Observation :
- description : String
- category : String
- value : Float

Procedure :
- code : String
- description : String

Most properties are string values, the only exceptions being the `healthcare_expenses` and `value` properties, which are encoded as floats and the temporal properties which are encoded as DateTime objects. The `HAS_ENCOUNTER` relationship from the patient to the encounters as well as the `HAS_`*`ENTITY`* relationships from an encounter to its associated medical entities, where *`ENTITY`* is the name of the associated entity have also been omitted from the simplified representation in Figure 4.1 above, but are also part of the patient journey model.

4.2. Dataset Creation

This section provides an overview of the process of creating the dataset and the data contained within it.

4.2.1. Synthea: Synthetic Patient Generator

"Synthea is an open-source, synthetic patient generator that models the medical history of synthetic patients." [37] It was first proposed in 2017 by Walonoski et al. [38] and has since expanded its capabilities to model a wide range of patient histories. It independently models the life of each patient and their EHR from birth (to death).

It does this using modules from its own *Generic Module Framework*, where each module is used to model events that can occur in a patient's life, such as encounters, conditions, medications, and procedures, stating that these statistical modules "are informed by clinicians and real-world statistics collected by the CDC, NIH, and other research sources." [37] An in-depth description of the framework and all available modules can be found in the official documentation [39].

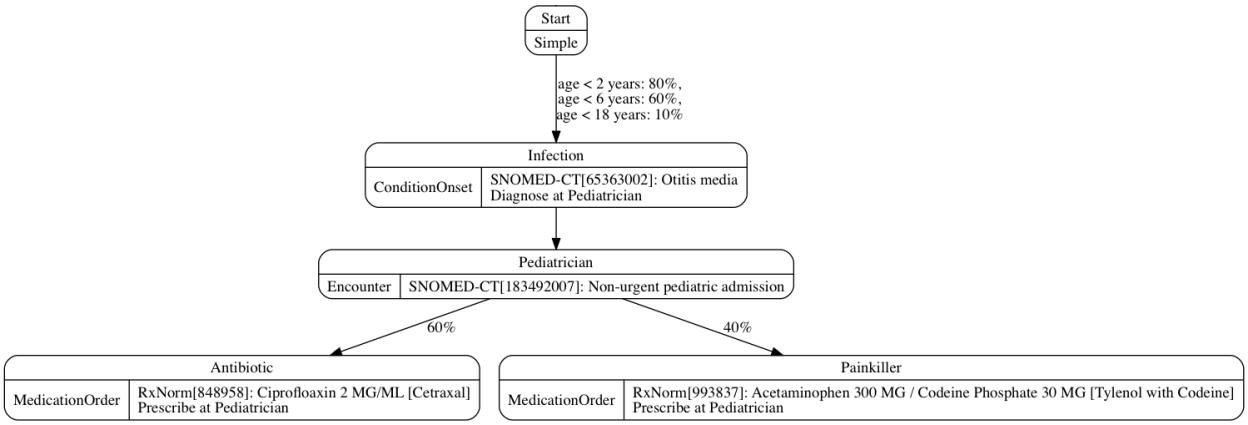


Figure 4.2.: Example of a Synthea Module [37].

It was chosen for this project because it allows for the user-friendly generation of large-scale, realistic - yet synthetic - patient data, which makes it useful for creating a dataset that can be shared and used for research purposes without violating patient privacy.

4.2.2. Using Synthea

Synthea is implemented in Java and can be run either from a packaged .jar file, as a Docker container, or built from source. The latter approach was chosen for this project, as it allowed for easier configuration, by modifying the `synthea.properties` file, which was adapted from an earlier version used in a different project [40] by Neo4j. It was updated to work with the latest version of Synthea and is configured to generate CSV files for the dataset which can be more easily imported into Neo4j than the default FHIR format.

The setup of Synthea was done on a Windows Subsystem for Linux (WSL) running Debian 11 (bullseye) on a local machine, which had access to 12GB of RAM, with an Intel i7-10510U CPU with a base clock rate of 1.8GHz. A Linux system was used because it was easier to set up and run Synthea on it than on Windows, as it is primarily developed for Unix-like systems.

Besides excluding information not consistent with the data model defined in 4.1 the default configuration was used. While it was considered to generate a dataset whose patient population is

more representative of German demographic data, it was decided against it. Patient demographics like names and addresses could have been adjusted to reflect German characteristics, however, the underlying statistical models are based primarily on U.S. healthcare data. Therefore, such surface-level changes would not have created a truly representative German patient population. The full configuration file is too extensive to be included here, but has been included in the supplementary materials submitted alongside this thesis.

To build and run Synthea, the following commands were executed in the project directory:

```
./gradlew build
./run_synthea -p $patient_count
```

Where `patient_count` specified the number of *living* patients to generate, which was set to 10, 100, 1000, and 10000 respectively. Generating Datasets of four different sizes, in order to analyze the scalability of the proposed embedding methods.

Since Synthea simulates patients' whole lives until the population reaches the amount specified in the census data it uses to model demographics, where only living people got counted, but some patients will have conditions causing their death before the population reaches that amount, it continues generating patients until the specified amount of living patients is reached. This leads to the number of generated patients being higher than the amount specified in the generation command. However, since medical outcomes causing death are relevant for generating realistic data and the number of dead patients scales with the number of living patients, the dead patients are included in the dataset.

The generated datasets were of sizes 10 (0 dead), 122 (22 dead), 1127 (127 dead), and 11618 (1618 dead), which means that the average death rate was 10.81% with a standard deviation of 6.69%. For the purpose of cleaner notation, however, the datasets will be referred to as the number of living patients they contain.

4.2.3. Creating Graph Representation

For the creation of the graph representation of the dataset, the generated CSV files were used to create nodes for patients, encounters, providers, conditions, medications, procedures, and observations, as well as the relationships between them according to the patient journey model defined in section 4.1. To do this, a modified version of the "pyingest" [41] script `ingest.py` from Neo4j was used, which works as follows:

1. It loads a `config.yaml` file that specifies the target database's connection details, the paths to the CSV files, and Cypher queries for each file, specifying how to create the nodes and relationships, as well as pre- and post-ingest queries.

2. It runs the specified pre-ingest queries. This was used to create indices for some of the nodes' properties to speed up the ingest process.

3. It iterates over the CSV files specified in the config file, loading them into memory and then adding them into them to the Neo4j database in chunks, by running the specified Cypher queries.

4. It runs the specified post-ingest queries. This was used to create `NEXT` relationships between encounters of the same patient, as well as the `FIRST` relationship between the patient and their first encounter.

The full modified script, as well as the `config.yaml` were submitted in the supplementary materials alongside this thesis and further explanation is omitted here, since beyond the specified steps the sake of reproducibility, it is not further relevant to the thesis.

4.3. Dataset Overview

This section analyzes the 10,000-patient dataset's characteristics and distributions to better understand common patterns and inform the interpretation of patient embeddings.

4.3.1. General Information

The dataset's demographic distribution is visualized in Figure 4.3, showing a structure typical of developed countries.

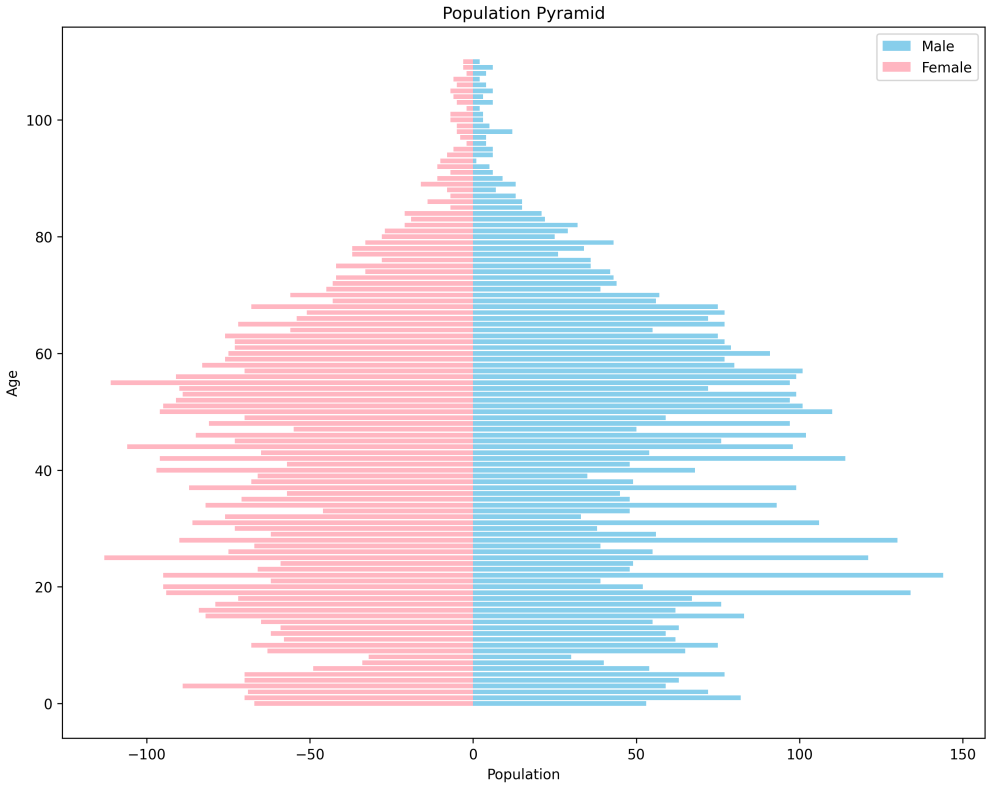


Figure 4.3.: Population Pyramid of the 10000 Patient Dataset.

The dataset comprises 6,937,466 nodes, including 304 conditions, 361 drugs, 400 procedures, and 6,071,300 observations. Due to computational constraints, observation nodes were excluded from further analysis. The prevalence of different medical entities provides insight into patient similarities.

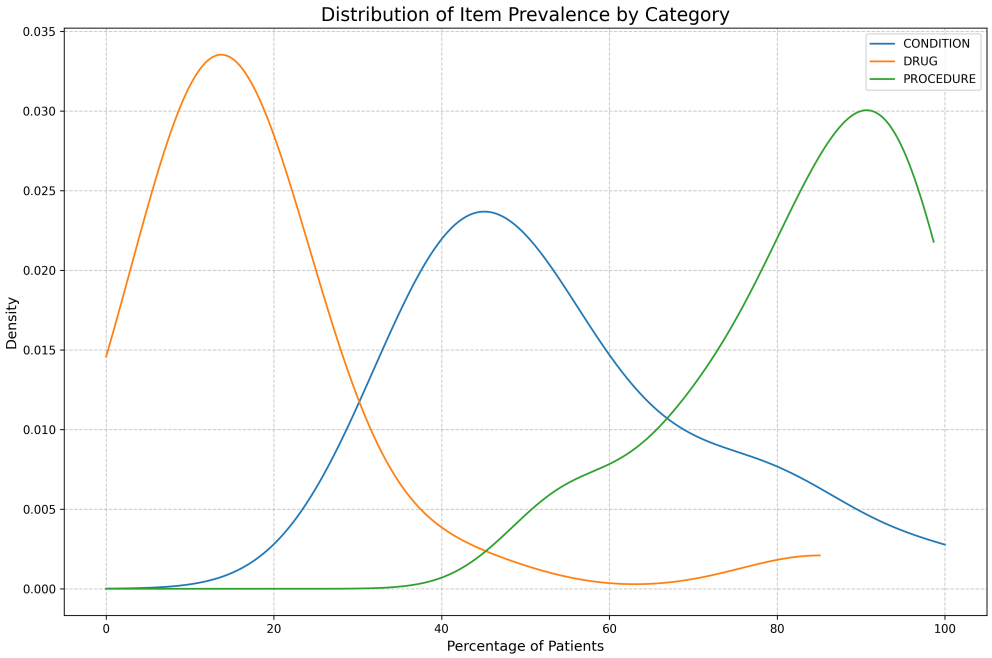


Figure 4.4.: Prevalence Distribution of Conditions, Drugs, and Procedures.

As shown in Figure 4.4, drugs tend to be rare and unique to patients, while conditions and procedures are more commonly shared across patients. This distribution suggests that drug-based embeddings may better capture individual patient characteristics, while condition and procedure-based embeddings may better represent shared health patterns.

4.3.2. Temporal Aspects

Temporal patterns in the data encompass patient age during encounters, inter-encounter intervals, and seasonal variations, all of which may influence embedding representations. The encounter rates shown in Figure 4.5 reveal that while medical visits increase with age, the patient population decreases, reflecting natural mortality patterns in the synthetic data generation. It also shows spikes in the number of patients having encounters at certain ages, most likely caused by the statistical models used in Synthea genearting regular check-ups at those ages.

Inter-encounter distances demonstrate significant variance across age groups, with particularly notable differences in younger patients. As shown in Figure 4.6, the intervals between encounters decrease with age, potentially encoding both patient age and health condition severity in the temporal patterns.

Beyond age-related patterns, the data also exhibits seasonal variations. Figure 4.7 shows how different types of encounters vary by month, suggesting cyclical patterns that may influence patient similarities in the resulting embeddings.

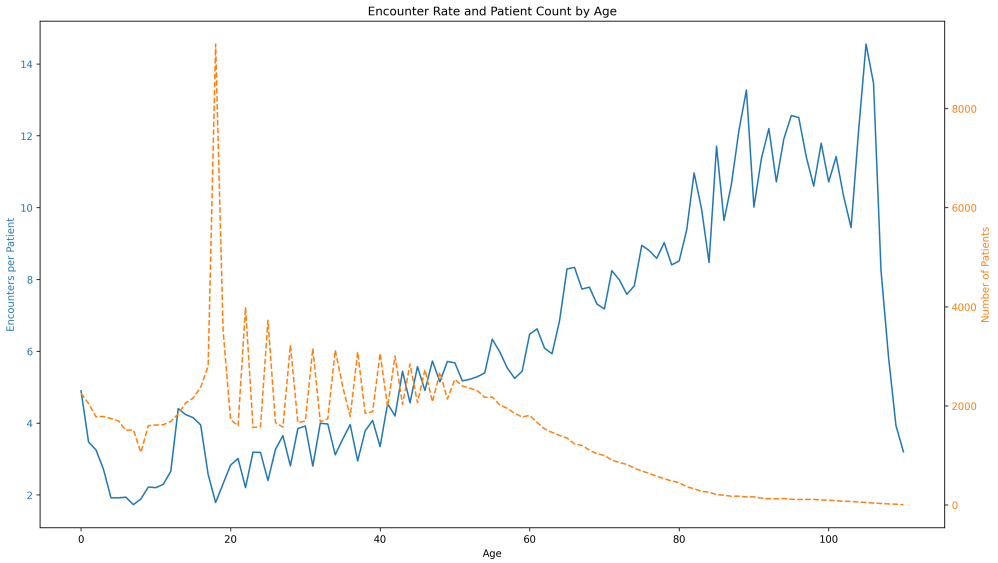


Figure 4.5.: Encounters Rate by Age.

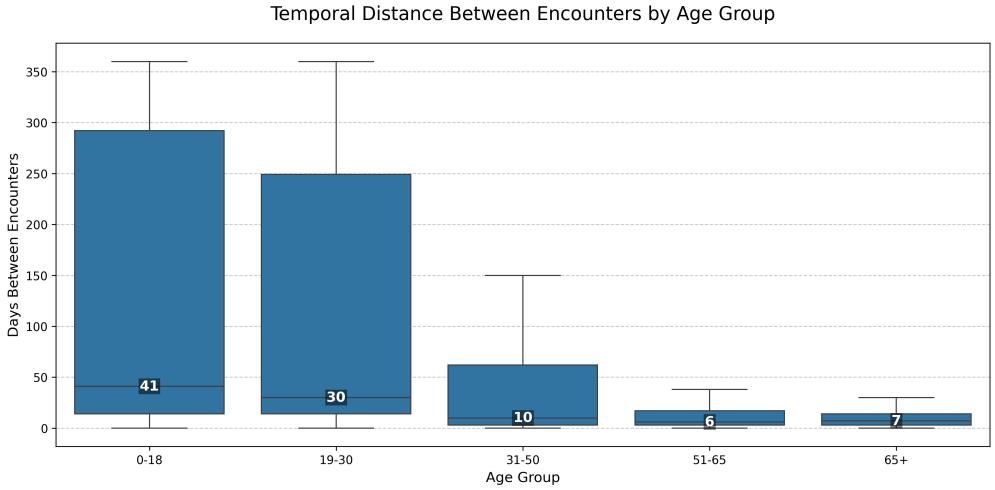


Figure 4.6.: Inter-Encounter Distances by Age Group.

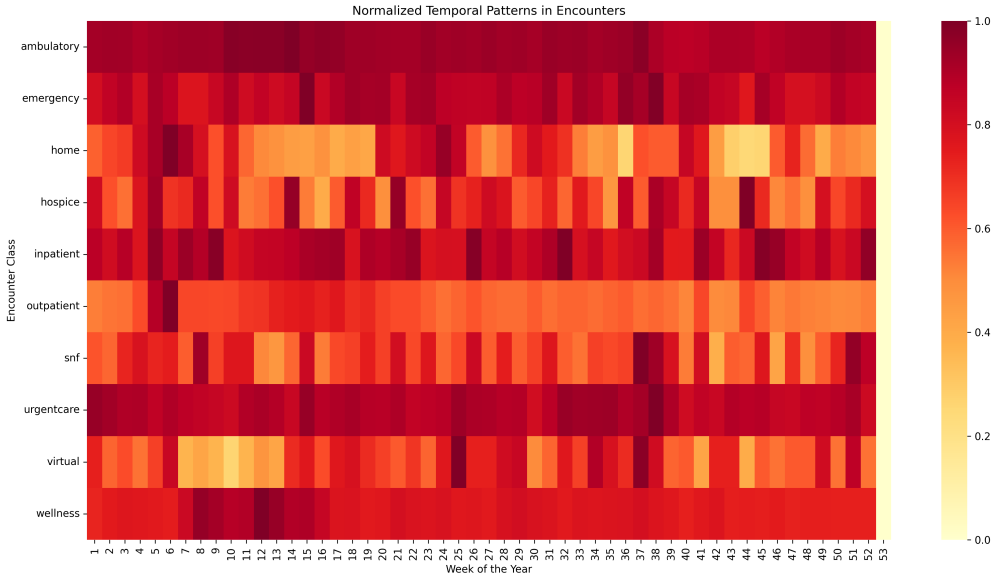


Figure 4.7.: Normalized Heatmap of Encounters by Month and Encounter Type.

5. Design and Implementation

This chapter details the implementation of a patient journey analysis system that transforms graph-structured clinical data into numerical embeddings. Using Neo4j's FastRP algorithm and custom encoding methods, the system generates patient-level representations that can be explored through an interactive Jupyter notebook.

5.1. Design

In order to leverage Neo4j's FastRP implementation, the graphs properties are encoded in ways allowing more meaningful node embeddings to be generated, which are then aggregated into a single graph level embedding for each patient. The embeddings are then clustered, dimensionally reduced and visualized.

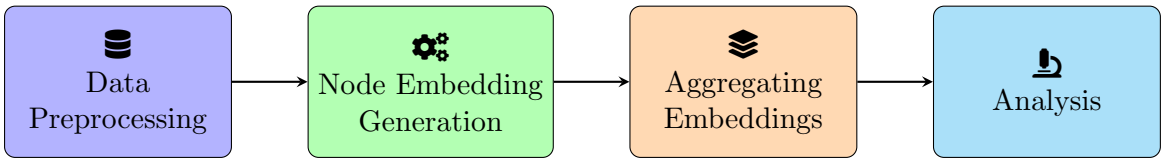


Figure 5.1.: Flowchart of the Proposed Methodology.

The entire workflow has been implemented as an interactive and extendable Jupyter notebook, submitted alongside this thesis, allowing for easy extension of the workflow and exploration of the data through the use of easily customizable embedding generation, as well as a custom visualization of the patient journeys within the notebook.

5.2. Implementation

This section describes the technical implementation details of the data preprocessing, embedding generation, and analysis workflows.

5.2.1. Data Preprocessing

Since FastRP requires numerical input, string properties in the graph must be encoded appropriately. Two methods were implemented: one-hot encodings and sentence transformer based encodings.

5.2.1.1. Encoding String Data

As described in section 4.1, most properties of the nodes are strings. Since FastRP can only utilize numerical properties, it is necessary to encode these strings in a meaningful way. The algorithms optimized implementation also requires that every node in the projected graph has a value for every encoded property and, in the case of arrays of scalar values, that the arrays are of the same length on all nodes.

To allow for the flexible selection of properties to encode, while still ensuring these constraints are met in all cases an interactive widget was implemented in the notebook. Using an APOC procedure, seen below in Listing 5.1, the nodes, their properties, and the types of these properties are fetched from the database.

```
CALL apoc.meta.nodeTypeProperties()
YIELD nodeType, propertyName, propertyTypes
RETURN nodeType AS label, propertyName AS prop, propertyTypes[0] AS valueType
```

Listing 5.1: APOC Query for Fetching the Available Node Types and Their Properties.

Then a selection of the node types and their string properties, as well as which encoding string type to use can be made. There are two implemented ways of encoding the string properties.

A naive method of encoding the string properties is to use **one-hot encodings**. To do so, for every selected property a list of distinct values over all the nodes with the selected labels is built. Then, the one-hot encoding of these values is added to the nodes inside the database in batches using a one-hot encoding function implemented within GDS.

```
MATCH (n)
WHERE ANY(label IN labels(n) WHERE label IN [{target_labels_str}])
AND n.`{prop}OneHot` IS NULL
WITH n LIMIT {batch_size}
WITH n, [toString(n.`{prop}`)] AS propValues
WITH n, gds.alpha.ml.oneHotEncoding($distinctValues, propValues) AS oneHotVector
SET n.`{prop}OneHot` = oneHotVector
RETURN COUNT(n) AS updated_nodes
```

Listing 5.2: One-Hot Encoding the String Properties.

Note that this query is run using the GDS python client and uses f-strings to dynamically insert the selected properties and labels as python variables inside curly braces in addition to the parameter denoted by the dollar sign in the query. The GDS `oneHotEncoding` function creates an array of zeros with the length of the amount of distinct values and sets the value at the index of the property value to 1. If the property is not present on the node, the encoding is an array of zeros. All selected labels are also encoded and added as a property to the corrresponding nodes.

This is a relatively fast way of encoding the string properties but it doesn't encode any semantic similarity between values. It also has the downside of the length of the arrays growing with the number of distinct values, which can be a problem for larger datasets.

These problems can be mitigated by using a **sentence transformer** model to encode the string properties. This is done by iterating over all nodes with the selected labels and encoding the property values with the *all-MiniLM-L6-v2* model from the "Sentence Transformers" [42] library, saving each encoded value into a dictionary to avoid encoding the same value multiple times. These encodings are then saved to the database in batches, writing an array of zeros, the length of the embedding size, for nodes that don't have the corresponding string property.

```
for prop_name in selected_properties:
    prop_value = row.get(prop_name)
    prop_encoded_name = f"{prop_name}SentTransf"

    if pd.notna(prop_value):
        if prop_value in value_embedding_map[prop_name]:
            embedding = value_embedding_map[prop_name][prop_value]
        else:
            embedding = model.encode(prop_value)
            value_embedding_map[prop_name][prop_value] = embedding
    else:
        embedding = np.zeros(EMBEDDING_DIM)
```

Listing 5.3: Encoding the String Properties with Sentence Transformer.

Both methods provide a deterministic way of encoding string properties, while ensuring they are present on all nodes in the projected graph and the arrays are of the same length on all nodes. Since FastRP linearly combines the randomly sampled vectors with the property arrays, arrays of zeroes do not influence the embeddings in any way.

5.2.1.2. Encoding Temporal Aspects

Since temporal information in the data is of particular interest but FastRP can't natively handle DateTime properties, these also need to be encoded in a way suitable for the algorithm. The focus being encoding the information contained in the `start` and `end` properties of the nodes, which can also be used to calculate the duration between encounters.

The timestamps are encoded using a pre-trained version of the **time2vec** layer described in section 2.2.4 called "Date2Vec" [43], which was trained on a next day prediction task and a date reconstruction task. First, all the `start` and `end` values of the nodes in the database are extracted. They are then converted to the input format of Date2Vec, passed through it and written back to the database in batches, similar to the sentence transformer based encodings.

The temporal distances between encounters are encoded as **edge weights** of the `NEXT` edges between them using a cypher query. The same is done with the distances between the `birthdate` of the patient and the encounters, as a measure of how old the patient was at the time of a given encounter. Since FastRP also requires the selected `relationshipWeightProperty` to be present on all projected edges, which is why it is set to 1 for all other relationships.

5.2.2. Embedding Generation

In section 2.5.2 many different hyperparameters of FastRP have been presented. In addition to these there also exist choices regarding which nodes, relationships, properties, and what way of representing them to use during node embedding generation, embeddings of which type of nodes to aggregate into graph level representations, whether or not to boost the importance of some nodes during the aggregation process, and if so by how much.

To enable faster iterations and flexibility with regard to data models in generating and exploring the embeddings, as well as ensuring that the selected parameters are fit to be used within the constraints of FastRP all of these options, with the exception of configuring the boosting, have been implemented as interactive and adaptable widgets within the notebook.

Selecting node types, relationships types, and properties during the embedding generation is all connected, since only properties present on the selected node types should be able to be selected, all selected node types should be reachable from the patient node, and only relationships with types that connect nodes of the selected node types should be selected for FastRP to work.

That is why the APOC procedure already presented in 5.1, together with a query containing information about the source and node types of relationship types, is used to build an adjacency list representation of the graphs schema and a dictionary of which node types have which properties.

These are then used to dynamically adjust the possible choice of properties, making sure that only those properties are able to be selected, which are present on all selected node types in the selected string encoding format, and that all nodes of the selected types are reachable through relationships of the selected types. The widget interface for this is shown in Figure 5.2.

Using the selected node types, another interactive widget provides the option of selecting which **node types to aggregate** into graph level embeddings, as well as the option to concatenate the temporal encodings to the node embeddings.

Through **defining a boost map** according to the schema presented in Listing 5.4, the importance of nodes of certain types or with certain property values during the aggregation can be increased.

```
boost_factors = {
    'NodeTypeName': {
        '_type': 1.1,
        'propertyName': {
            'propertyValue': 10
```

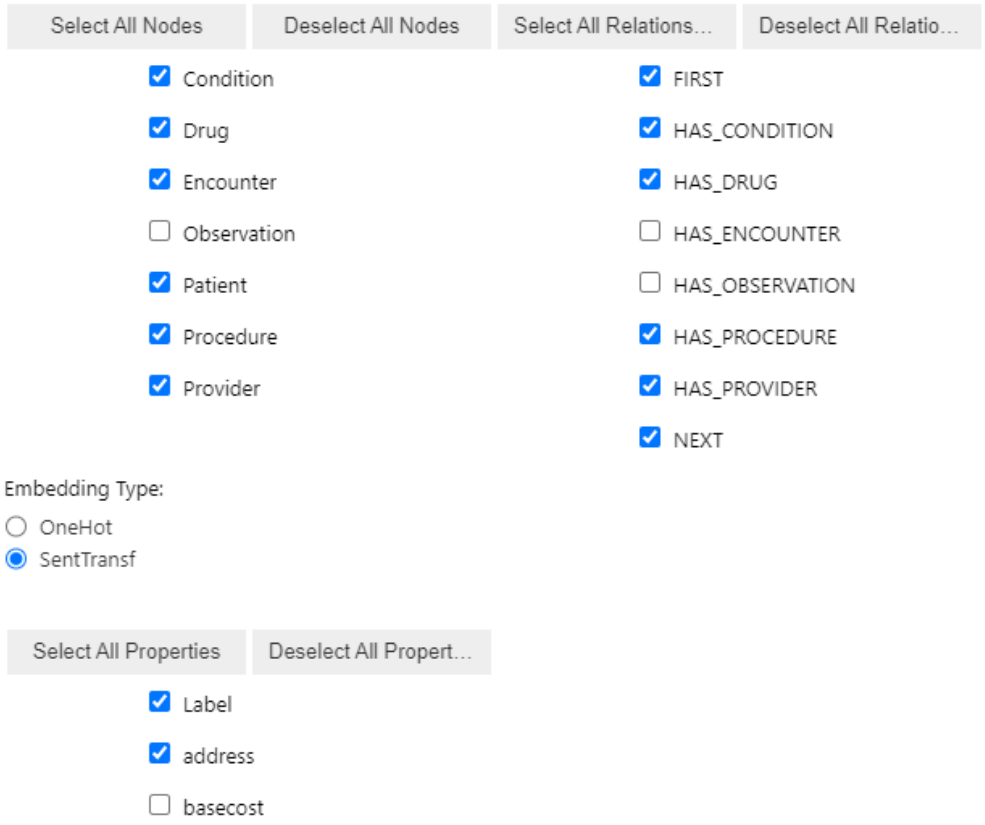


Figure 5.2.: Widget for Selecting Node/Relationship Types and Properties.

```
        }
    }
}
```

Listing 5.4: The Boost Map Schema for Aggregation.

Where the `_type` key is used to boost the importance of all nodes of a certain type and the `attributeName` and `attributeValue` keys are used to boost the importance of nodes with a certain property value.

Lastly **setting the hyperparameters for FastRP** is also done through a widget, making sure that, if no properties are selected, the property ratio and node self influence are set to 0.0.

The hyperparameters selected in this work are:

- **embeddingDimension**: 256
- **propertyRatio**: 1.0
- **iterationWeights**: [1.0, 1.0, 1.0, 1.0]
- **nodeSelfInfluence**: 1.0
- **relationshipWeightProperty**: 'weight'

where `propertyRatio` and `nodeSelfInfluence` were only set to 1.0 if properties were selected for the tested embeddings and `relationshipWeightProperty` was only set to *weight* if temporal edge weights were used. Additionally a seed for the randomness inside FastRP was specified to ensure reproducibility of the results.

These values were selected based on recommendations inside the FastRP documentation and showed sufficiently good results in testing, providing a good balance between speed and quality of the embeddings, although they could be further optimized through hyperparameter tuning, given a more concrete task to optimize for.

All of the previous selections are then used to dynamically build the query for **generating the node embeddings** using a python f-string, which is then executed in batches over all patients in the database using the GDS python client.

```
UNWIND $batch_patient_ids AS patientId
MATCH (p:Patient {{id: patientId}})
WITH p
CALL {{
    WITH p
    MATCH path = (p)-[*0..]->(n)
    WHERE ALL(r IN relationships(path) WHERE type(r) IN {list(selected_relationships)})
    WITH p, COLLECT(DISTINCT n) AS nodesInPath
    WITH p, nodesInPath, [n IN nodesInPath | id(n)] + id(p) AS nodeIds

    CALL gds.graph.project.cypher(
        'patientJourney_' + p.id,
        'MATCH (n) WHERE id(n) IN $nodeIds
            RETURN id(n) AS id,
                labels(n) AS labels{',' if feature_properties else ''}
                {", ".join([f"n.{prop} AS {prop}" for prop in feature_properties])}',
        'MATCH (n)-[r]-(m)
            WHERE id(n) IN $nodeIds AND id(m) IN $nodeIds
            RETURN id(n) AS source, id(m) AS target{project_match_end}',
        {{
            parameters: {{
                nodeIds: nodeIds
            }}
        }}
    )
    YIELD graphName

    CALL gds.fastRP.stream(
        graphName,
        {{
```

```
31              embeddingDimension: {embedding_dimension},
32              featureProperties: { list ( feature_properties )},
33              propertyRatio: {property_ratio},
34              randomSeed: {random_seed},
35              iterationWeights: { iteration_weights },
36              nodeSelfInfluence: { node_self_influence },
37              relationshipWeightProperty: {relationship_weight_property}
38          }}
39      )
40      YIELD nodeId, embedding
41      MATCH (n:{"|".join(output_types)}) WHERE id(n) = nodeId
42      WITH p, graphName, n, embedding{concat_case_statement}{boost_case_statement}
43      WITH p, graphName, {output_statement}
44
45      CALL gds.graph.drop(graphName) YIELD graphName AS droppedGraph
46
47      RETURN p.id AS patientId, nodeEmbeddings
48  }}
49  RETURN patientId, nodeEmbeddings
```

Listing 5.5: Generating the Node Embeddings using FastRP.

First, in lines 4–26, an in-memory projection of the patient journey subgraph containing only the selected node and relationships types, as well as the selected properties, is created. This projected graph is then passed to the FastRP algorithm called in lines 28–40, which generates the node embeddings for the projected graph using the selected hyperparameters.

In lines 41–43 first only the nodes of the types selected for aggregation are matched, then the node embeddings for the patient are grouped into a list, optionally concatenating the property containing the Date2Vec temporal encoding to them or adding an additional boost factor for every node embedding if the boost map was set.

Finally the projected graph is dropped from the database and the embeddings are returned together with the patient id. As the query is run from the GDS python client, the data is returned as pandas dataframes, which are then concatenated into a single dataframes containing the patient ids and their corresponding node embeddings.

5.2.3. Aggregation

Depending on whether a boost map was set or not, the node embeddings are aggregated into a single graph level embedding for each patient by either taking the element-wise mean of the embeddings or by using a weighted mean based on the boost factors.

While this is a simple way of aggregating the node embeddings into graph level representations, it is a computationally inexpensive and commonly used method of doing so [7]. A brief discussion of potentially more effective but more complex methods can be found in chapter 7.

5.2.4. Analysis

In order to gain insights into the structure of the generated embeddings and how they represent the patient journeys in the embedding space, they are clustered, dimensionally reduced and visualized inside a two dimensional and a three dimensional plot. The latter of which allows for the interactive display of the k-th nearest neighbors of a given embedding and the selection of one or more embeddings to view the corresponding patient journeys from within the notebook.

For **clustering** and **dimensionality reduction**, widgets are provided for selecting the hyperparameters of the HDBSCAN and UMAP algorithms. In chapter 6, different options for HDBSCANs hyperparameters are discussed for different embedding strategies. The UMAP hyperparameters can be freely set to explore either more local or more global structures of the data.

The **interactive two-dimensional plot** is implemented using the `plotly` library for python. Using a dropdown menu a k value can be selected and the k-th nearest neighbors of an embedding can be highlighted by clicking on it. An example is shown in Figure 5.3, where the first nearest neighbor of the selected embedding is highlighted.

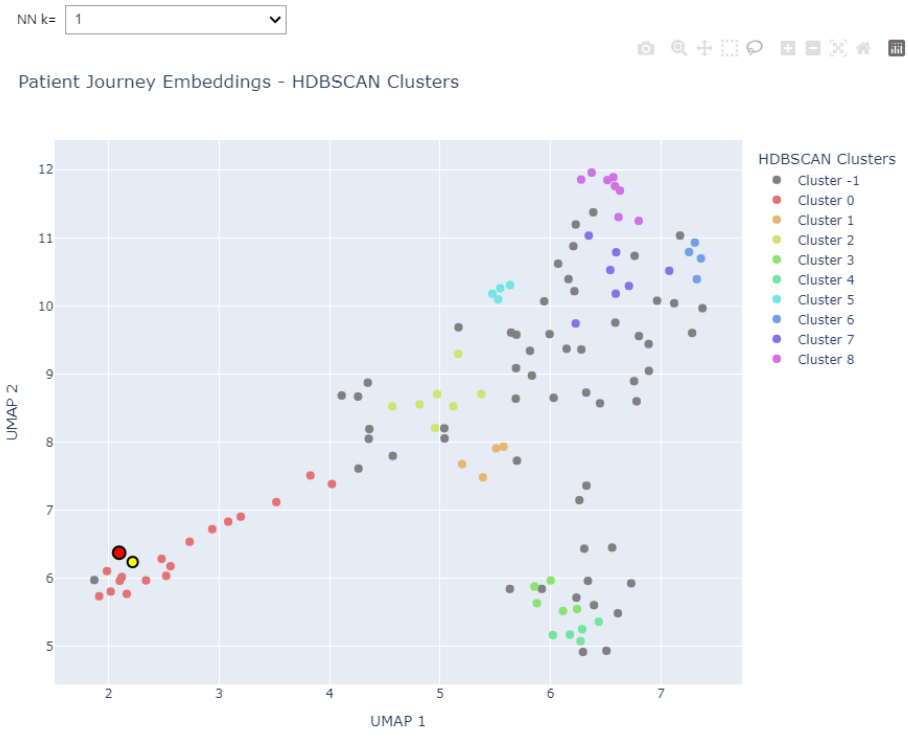


Figure 5.3.: Viewing the Nearest Neighbor of an Embedding.

In Figure 5.4 both the patients embeddings and that of their nearest neighbor have been selected by encircling them in the plot.

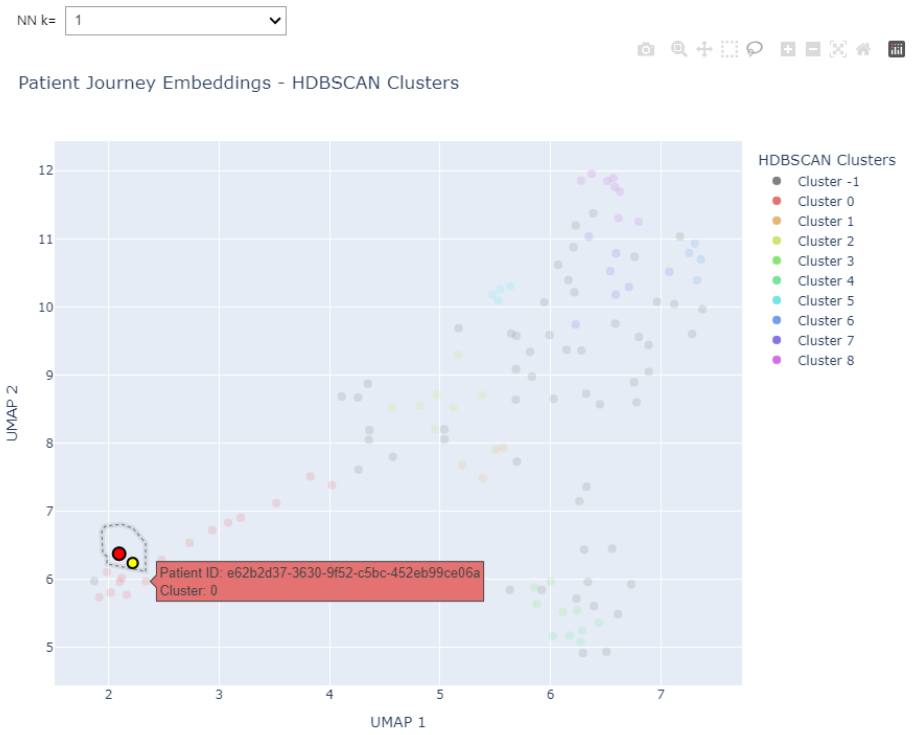


Figure 5.4.: Selecting a Patient for Journey Visualization.

Upon selecting one or more embeddings in the two-dimensional plot, a cypher query for fetching the corresponding patient journeys can be generated and used for visualizing the associated patients' journeys in the notebook. This functionality is implemented through the use of the `yfiles_jupyter_graphs_for_neo4j` package, which has been configured to provide an intuitive visualization of the journey based on the patient journey model. An example of the visualization of the journeys of the patients selected in Figure 5.4 is shown in Figure 5.5.

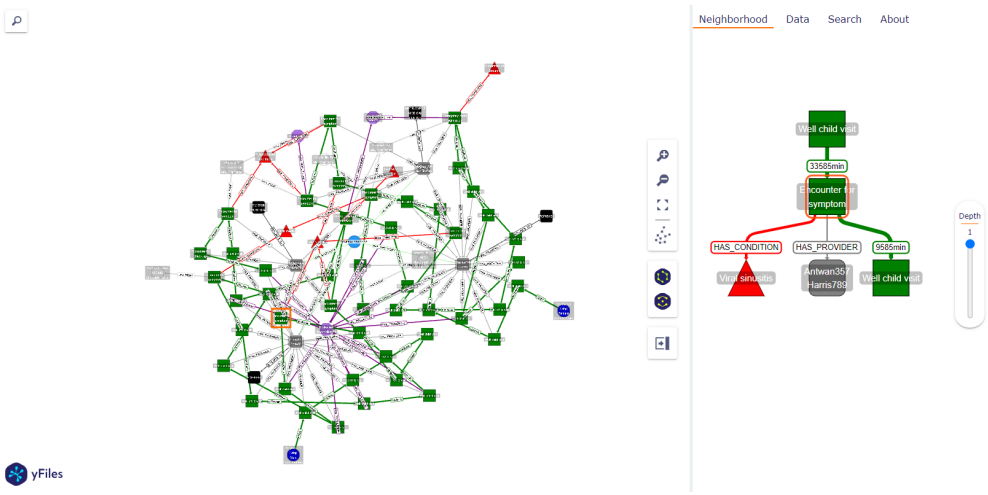


Figure 5.5.: Visualizing the Selected Patient Journey with yFiles.

6. Evaluation and Discussion

Graph level embeddings are complex and abstract objects, which makes them difficult to evaluate when lacking a ground truth to compare them to. Even though a manual inspection can provide some insights, these can hardly be considered robust evidence about the quality of generated embeddings, as they are not quantifiable.

To make informed claims about the validity of the methods proposed in this work and the quality of the embeddings generated with them it is crucial to evaluate whether they have the ability to capture different characteristics of patient journeys, namely their topological, semantic, and temporal aspects and if structures in the data can be made apparent through clustering.

First, the effectiveness of the embeddings in capturing differences with regard to topological, semantic, and temporal differences is evaluated by constructing a ground truth for these categories. Based on the results, the most promising candidates of the embedding methods are evaluated with respect to their scalability and performance in a clustering task against a baseline method. Finally the results are discussed and key findings are summarized.

6.1. Effectiveness Evaluation

For the purpose of constructing the discussed ground truths, two patient journeys were extracted from the 10000 patient dataset by use of an APOC function, seen in Listing 6.1. The smaller of which will only be used for the evaluation of the topological differences, while the larger one will be used for the evaluation of the semantic and temporal differences.

```
MATCH (p:Patient {id: $patient_id})
CALL apoc.path.subgraphAll(p, {
 relationshipFilter :  'HAS_CONDITION>|HAS_ENCOUNTER>|HAS_DRUG>|
     HAS_PROVIDER>|NEXT>|FIRST>|HAS_PROCEDURE>',
minLevel: 0,
maxLevel: -1
})
YIELD nodes, relationships
RETURN nodes, relationships
```

Listing 6.1: APOC Procedure for Efficiently Extracting Patient Journeys.

They were then saved in-memory as a python variable, seen in Listing 6.2, by storing their nodes' ids, labels, and associated properties, filtering out all properties that were added during the preprocessing done on the source database. For the relationships the ids of the source and target nodes and their types were stored.

```
nodes = []
edges = []
for node in nodes_result:
    nodes.append(
        {
            'old_element_id': node.element_id,
            'label' : "".join(node.labels),
            'properties': {
                key: node[key] for key in node.keys() if not (key.endswith('OneHot') or key.
                    endswith('SentTransf') or key.endswith('D2V'))
            }
        }
    )

for relationship in relationships_result :
    edges.append(
        {
            'start_id' : relationship.start_node.element_id,
            'end_id' : relationship.end_node.element_id,
            'type' : relationship.type
        }
    )

return {'nodes' : nodes, 'edges' : edges}
```

Listing 6.2: Python Code to Save Patient Journeys in Custom Representation.

Note that the nodes ids are denoted as *old_element_id*, as they are the ids of the nodes in the source database. When the nodes of graphs in this format are written to the evaluation database, a mapping of the old ids to their newly assigned ids is created and used to correctly assign the relationships to the nodes in the new database.

Storing the graphs in this format allows for modifications to be made in a fast and easy to implement way, which is required for creating the ground truths for the different categories. For each category a set of modifications is applied to the graph, altering it in different ways with regard to the category in question, while keeping its structure with regard to the other categories intact. Then the cosine similarity between the embeddings of the different versions of the graph is calculated for different embedding methods and it is evaluated whether the embedding methods are able to capture the differences, if so how much of an influence they ascribe to the different categories, and whether that is consistent with how large the modifications were.

6.1.1. Topological Differences

First it was evaluated whether the aggregated embeddings capture topological differences between a smaller and a larger patient journey. Both extracted graphs were transferred to the evaluation database as is and the embeddings were generated with:

NP - No properties.

STP - Sentence transformer encoded properties.

STSP - Sentence transformer encoded properties and scalar properties.

Besides the topological differences the graphs also differed moderately in the string properties encoded through the sentence transformer. The reason for including the STP and STSP embeddings was not to evaluate how well the embeddings capture the string properties, but rather to evaluate if the topological differences are still captured when these different aspects are included.

Metric	NP	STP	STSP
Similarity	0.1199	0.7832	0.9947

Table 6.1.: Comparison of Embedding Methods for Capturing Topological Differences.

It can be seen that the topological difference captured by the node level embeddings still holds when the embeddings are aggregated to the graph level, even when the string properties are included.

However, including `basecost`, the only scalar property left after excluding the `Observation` nodes due to their mass, in the STSP embeddings leads to the topological being completely overshadowed. This is consistent with the way FastRP constructs the initial sparse vectors of nodes when using a `propertyRatio` 1.0 as a linear combination of sparse vectors for each scalar property and each index of the arrays of scalar values. For over all the nodes in the evaluated graphs, `basecost` has a mean of 135.34 with a standard deviation of 78.22, while the scalar values in the arrays generated by the sentence transformer are mostly distributed around 0. This causes the sparse vector generated for the `basecost` to be orders of magnitude more influential than the other properties. For this reason `basecost` was not further considered as a relevant property during the evaluation.

6.1.2. String Property Differences

In order to test the ability of the embeddings to capture semantic differences between different graphs and the ability of the boosting strategy to influence the embeddings, the larger of the two patient journeys was modified with regard to its string properties. The following transformations were applied:

No Properties (NP) - All properties were removed.

Test 1 (T1) - All string properties were replaced with "TEST_PROP".

Test 2 (T2) - All string properties were replaced with "TEST_PROP`ij`" (where i corresponds to the i-th node and j to the j-th property). Additionally, a property `boostprop` with value "SAME_i" was added to each i-th node.

Test 2 Boost (T2_B) - Same as T2, but with `boostprop` set to "THISISDIFFERENT" for the first and last encounter.

The cosine similarity of the sentence transformer encodings is between 0.7 and 0.85 for "TEST_PROP" and "TEST_PROP`ij`", between 0.6 and 0.8 for "SAME_i" with different i's, and between 0.1 and 0.3 for "THISISDIFFERENT" and "SAME_i" with different i's, meaning that the former two similarities are encoded as much more semantically similar than the latter.

The modified versions of the graph were then written into the evaluation database and the embeddings were generated using:

NP - No Properties.

OHP - One-hot encoded properties.

STP - Sentence transformer encoded properties.

STP_TB - Sentence transformer encoded properties with boosting encounter nodes with a factor of 10.

STS_VB - Sentence transformer encoded properties with boosting encounter nodes with the value "THISISDIFFERENT" with a factor of 10.

Allowing for a comparison between how well the one-hot encoding and the sentence transformer strategies are able to capture differences in the string properties against a baseline of no properties. Additionally, the boosting strategy for node types as well as values together with the sentence transformer is evaluated.

Method	NP-T1	NP-T2	NP-T2_B	T1-T2	T1-T2_B	T2-T2_B
NP	-0.0022	-0.0677	-0.0652	-0.0598	0.0437	0.0458
OHP	0.3605	0.8679	0.8677	0.3615	0.3608	0.9996
STP	0.4656	0.4727	0.4730	0.8441	0.8444	1.0000
STP_TB	0.4631	0.4723	0.4727	0.8299	0.8303	0.9999
STS_VB	0.4656	0.4727	0.4803	0.8441	0.8381	0.9729

Table 6.2.: Comparison of Embedding Methods for Capturing Seminatic Differences.

The results show distinct differences between one-hot encoding and sentence transformer methods. One-hot encoding captures the uniqueness of T1 but fails to differentiate between NP, T2, and T2_B, as their initial embedding vectors become similarly random due to the summation of multiple sparse vectors, since for $n = i \times j$ different property values of T2 and T2_B n sparse vectors get generated. In contrast, the sentence transformer method successfully captures the semantic similarity between T1 and T2/T2_B while maintaining appropriate distance to NP, though it cannot differentiate between T2 and T2_B by itself.

Boosting all encounter node embeddings by a factor of 10 produces minimal changes, while targeted boosting of nodes with the "THISISDIFFERENT" value in `boostprop` effectively highlights these differences. The sentence transformer method proves more viable for capturing semantic similarities, while one-hot encoding, though potentially useful for categorical data, shows significant limitations.

The boosting technique, while effective for emphasizing specific values, is not considered for further evaluation.

6.1.3. Temporal Differences

To evaluate the ability of the embeddings to capture temporal aspects of the patient journeys, four transformations were applied to encounters of the larger journey and their directly associated nodes. Each transformation alters different temporal properties while maintaining others constant:

Uniform (UNI) - Sets the temporal distance between all encounters to their mean value, modifying relative distances while maintaining total journey duration.

Flipped (FLIP) - Reverses the order of encounters, preserving both relative and absolute distances but changing their sequence.

Absolute (ABS) - Shifts all encounters one year into the future, maintaining all relative temporal relationships while changing absolute timestamps.

Delta (DELT) - Multiplies the temporal distance between encounters by ten, expanding the total journey duration while preserving relative proportions.

These transformations allow for the isolation of specific temporal aspects: only UNI modifies relative distances between encounters, only DELT changes the total journey duration, only FLIP alters the encounter sequence, and only ABS shifts the absolute timestamps.

The methods of encoding temporal differences, of which all permutations were evaluated, were:

NT - No temporal encoding.

WGT - The precomputed weights on `NEXT` and `HAS_ENCOUNTER` relationships as relationship weights in FastRP.

D2Vprop - The Date2Vec encoded properties as feature properties in FastRP.

D2Vconc - The Date2Vec encoded property `start` concatenated to the corresponding node embedding vector.

For the evaluation of temporal characteristics, the sentence transformer method was used to encode a selected set of non-temporal string properties: address, category, class, code, description, ethnicity, gender, name, race, and specialty. These properties cover all relevant data from drug, condition, procedure, and provider nodes, while including only the most descriptive patient properties, as encoding all patient-specific properties would create unnecessary computational overhead without adding informational value.

The Table 6.3 shows the similarities of the modified journeys to the unaltered journey under each permutation of temporal encoding methods.

Method	UNI	FLIP	ABS	DELT
NT	1.0000	1.0000	1.0000	1.0000
EW	1.0000	1.0000	1.0000	1.0000
D2Vprop	0.5193	0.9629	0.9994	0.9486
D2Vconc	0.6273	0.8763	0.9999	0.8988
EW + D2Vprop	0.6874	0.9962	0.9994	0.9733
EW + D2Vconc	0.6195	0.8747	0.9999	0.8976
D2Vprop + D2Vconc	0.5193	0.9629	0.9994	0.9486
EW + D2Vprop + D2Vconc	0.6161	0.8746	0.9999	0.8975

Table 6.3.: Comparison of Embedding Methods for Capturing Temporal Differences

The results demonstrate that, while temporal edge weights failed to capture temporal differences, both methods of using the Date2Vec encodings (as feature properties and concatenated node embeddings) made it possible to distinguish between embeddings of the modified journeys and the unaltered journey. Notably, when both Date2Vec implementations were combined, the feature property version showed dominant influence.

With both versions the biggest change in similarity can be observed for UNI, which is the only transformation that changes the relative temporal distance between encounters. Together with the fact that the smallest change in similarity happened to ABS, which only changes the absolute temporal distance between encounters, this indicates that the Date2Vec encoded properties are mainly capturing the relative temporal distance between encounters This seems consistent with the design of the time2vec layer, which is designed to mainly learn periodic patterns and only has a single neuron for linear patterns. Despite that both versions still manage to capture the temporal differences for FLIP and DELT, which maintain the relative temporal distance between encounters. This might be due to both transformations changing the timestamps of the encounters and their directly associated nodes, in a way that is less linear than the shift by exactly one year by ABS, changing not just the year but also the months, days, hours, and minutes, and therefore a much larger part of the input to the pre-trained time2vec model.

Since both Date2Vec based temporal encoding methods seem to be able to capture the temporal differences, both will be used in the following evaluation steps together with the sentence transformer method for encoding string properties. Additionally for the clustering task a naive baseline journey embedding method and the sentence transformer method without any temporal encoding will be used as a comparison.

6.2. Baseline Method

The purpose of creating a baseline method is to have a simple and fast method, not involving any topological features, to compare the more complex and computationally expensive graph based methods to. In order for it to still be comparable to the other methods, it should contain some semantic and temporal information about the patient journey, while foregoing the topological information.

The method chosen for the baseline in this work extracts the 5 most common and 5 least common string property values for every property in the patient journey, calculates a sentence transformer encoding for each of them, and then averages them all together to get a single vector describing the journeys string properties. This vector is then concatenated with the Date2Vec encoded properties of the first and last encounter in the journey to provide it with some temporal information.

6.3. Scalability Evaluation

All times listed in this section pertain to the runtime of the specified processes within a jupyter notebook executed through the vscode jupyter extension on a machine with an Intel Core i7-10510U CPU with a base frequency of 1.8 GHz and 16 GB of RAM running Windows 11 and are provided in the format `hours:minutes:seconds`.

6.3.1. Preprocessing

As the embedding generation methods chosen for the clustering task rely heavily on preprocessing with the sentence transformer model and Date2Vec, the runtime of both methods on the four sizes of datasets was measured. For the purpose of comparison the time for generating and writing the one-hot encodings was also measured, resulting in Table 6.4 below.

Method	**10**	**100**	**1000**	**10000**
One-hot	00:00:06	00:00:14	00:02:33	01:34:22
Sentence transformer	00:02:27	00:06:15	00:52:51	09:13:55
Date2Vec	00:00:03	00:00:11	00:01:37	00:19:32

Table 6.4.: Runtime of Preprocessing Methods for Different Dataset Sizes.

When investigating the scaling behavior of the preprocessing methods, all methods exhibit super-linear scaling behavior as the dataset size increases, though to varying degrees. The one-hot encoding method shows the most dramatic super-linear scaling, with runtime increases growing from 2.3x to an extreme 37x for each 10-fold increase in dataset size. The sentence transformer method, while still super-linear, shows more moderate scaling with runtime increases ranging from 2.5x to 10.5x. Date2Vec also demonstrates super-linear scaling, with increases from 3.7x to 12.1x, though it processes significantly less data per sample than the sentence transformer method due to its simpler input requirements.

6.3.2. Embedding Generation

The runtime of the embedding generation methods was measured for all selected embedding methods on the four sizes of datasets, the results of which are shown in Table 6.5.

The generation speeds are very similar, due to the embedding of the sentence transformer encoded properties taking much longer than that of the Date2Vec based properties. This can be seen when

Method	10	100	1000	10000
STP + D2Vprop	00:00:15	00:01:36	00:15:47	11:41:20
STP + D2vconc	00:00:05	00:01:36	00:13:11	11:13:58
STP	00:00:04	00:00:57	00:12:40	10:51:00

Table 6.5.: Runtime of Embedding Generation Methods for Different Dataset Sizes.

comparing STP which only uses the former with the other methods and is explained by the discrepancy between the dimensionality of the vectors generated by the sentence transformer model (384) and Date2Vec (64), as well as the higher prevalence of string properties than date properties in the data.

6.4. Clustering Task

The clustering task was performed on embeddings generated from the 10000 patient dataset with each of the selected embedding methods. Each clusterings quality was evaluated using the DBCV index. To enable a fairer comparison a grid search was done for the HDBSCAN hyperparameters, finding the optimal values for each set of embeddings over the ranges:

min_cluster_size [10, 50, 100, 200, 300, 500]

min_samples [2, 5, 5, 10, 20, 50, 100, 200]

cluster_method [eom, leaf]

Additionally to the DBCV index, the hyperparameters (cluster selection method, minimum samples, minimum cluster size), the number of clusters, and the percentage of noise points were recorded for each clustering, to provide more insight into their structure.

Method	Sel.	Min. S.	Min. C. S.	n Clusters	% of Noise	DBCV Score
Baseline	eom	2	50	4	74.40%	0.2560
STP + D2Vprop	leaf	2	10	101	83.66%	0.1634
STP + D2Cconc	eom	2	200	2	77.67%	0.2233
STP	eom	20	10	2	4.87%	0.9513

Table 6.6.: Performance of Different Embedding Methods on Clustering Task.

The results in Table 6.6 show that, according to the DBCV score, the simple baseline method outperforms both methods that incorporate temporal properties, while using only the encoded string properties yields the highest score. Analysis of the optimal hyperparameters provides insight into these results.

For the method using encoded dates as features, optimal clustering is achieved with the `leaf` cluster selection method, resulting in numerous smaller, more homogeneous clusters, indicating that there might be many small clusters in the data that are hard to distinguish. The embeddings that concatenate temporal properties differ from the string-only embeddings only in their final 32 dimensions. This explains why they produce the same number of clusters as the string-only method (STP) - they likely discover clusters based on the same underlying distribution, but with added

noise from the temporal data. This noise appears to cause the algorithm to select only the densest parts of the clusters that were discovered in the string-only approach.

The lower performance of temporally-integrated embeddings, even compared to the baseline, suggests that incorporating temporal data creates a more uniform distribution of points in the embedding space, making distinct clusters harder to identify. However, this doesn't necessarily mean the embeddings fail to capture important data properties. Rather, it suggests that temporal information introduces natural variation in the data that, while potentially meaningful, makes traditional density-based clustering more challenging.

6.5. Summary and Discussion

It was discovered that FastRP node embeddings maintain their ability to capture topological differences when using mean aggregation, validating this approach for generating graph-level embeddings. Furthermore, weighted mean aggregation (boosting) proved to be an effective method for fine-tuning embeddings, potentially allowing for the incorporation of domain knowledge to adjust similarity scores.

The sentence transformer model "all-MiniLM-L6-v2" successfully captured semantic differences between graphs when encoding string properties, outperforming one-hot encoding in both effectiveness and scalability.

For temporal aspects, the pre-trained "Date2Vec" model successfully captured temporal differences between patients, with a notably stronger effect on relative time differences between encounters compared to absolute temporal shifts.

The clustering analysis revealed that embeddings based on string properties create denser point distributions in the embedding space, facilitating distinct cluster identification. However, incorporating temporal properties leads to a more uniform distribution, making cluster identification more challenging. This reduced clustering performance does not necessarily indicate lower quality embeddings overall.

The significant impact of temporal features on embedding similarity, particularly regarding relative encounter distances (see "UNI" in Table 6.3), combined with the high variance in encounter timing observed especially in younger patients (Figure 4.6), suggests that temporal patterns might be either too variable or too dominant in the embeddings for effective clustering. Despite this limitation in clustering performance, these embeddings accurately reflect temporal differences between patient journeys and may prove valuable for other applications, such as patient similarity search, though proper evaluation would require domain expertise to establish ground truth.

7. Conclusion and Outlook

7.1. Conclusion

This thesis explored approaches to generating graph level embeddings of temporal property graphs containing health data, utilizing Neo4j's Graph Data Science Library implementation of FastRP node embeddings. A synthetic patient dataset was created using Synthea and analyzed with focus on its temporal features to understand the information potentially encoded in the resulting embeddings. Various approaches for incorporating this information through preprocessing and aggregating node level embeddings into graph level representations were evaluated for effectiveness, scalability, and clustering performance. Additionally, an interactive notebook was developed to visualize embeddings and explore parameter effects, enabling intuitive visual analysis of the resulting graph level embeddings.

The results of the evaluation show that using a sentence transformer model to encode string information is an effective and scalable way of enabling the FastRP implementation in GDS to benefit from them. Mean aggregation proved viable for transforming node embeddings into graph level representations, with weighted aggregation offering opportunities for domain-knowledge-based tuning. A pre-trained time2vec layer successfully captured temporal differences in patient journeys, with relative encounter distances showing the strongest impact, followed by encounter order, temporal distance magnitude, and to a lesser extent, absolute temporal positions. While string property embeddings showed strong clustering performance compared to the baseline, adding temporal information prevented the identification of dense, well-separated clusters, likely due to high temporal variance in the dataset.

7.2. Outlook

As this thesis follows an *explorative* approach of generating graph level embeddings, it is not exhaustive in its trials of potentially meaningful approaches. Further, with more computational capacities different avenues of evaluation, which might lead insight into potential for domain specific tasks could be enabled.

Since encoding the temporal information shows promising results in effectiveness evaluation but clustering the data falls short in recognizing underlying structures, experimentation with the dimensionality and therefore influence on the graph level embedding of the vectors encoding the timestamps, as well as controlling the dataset for temporal variations could yield insight into what levels of influence of the temporal data is appropriate for embedding patient data.

With more computational resources, evaluating embeddings with on domain specific tasks, like emergency stay prediction or death prediction, would be possible by generating a larger dataset, assigning ground truth values to relevant outcomes and truncating them out of the journeys, then training a classifier. This would provide insight into the methods' viability in medical contexts and

enable training a custom time2vec layer alongside the classifier for optimized temporal representation.

Generally, while encoding information through pre-trained models (sentence transformers and time2vec), generating node-level embeddings, and aggregating them into graph-level representations makes efficient use of implemented functionalities and effectively encodes patient journey aspects, the approach using Neo4j's FastRP and mean aggregation remains inherently static. Optimizations are limited to choosing encoding methods, tuning hyperparameters, and varying aggregation methods, without practical ways to optimize embeddings for specific downstream tasks through end-to-end training. This limitation exists despite the much of the effectiveness of the presented embedding methods stemming from neural networks, which theoretically enables fine-tuning of individual components like the sentence transformer or time2vec models. While these components could be fine-tuned separately, the current architecture, makes it impractical to backpropagate loss from a classifier layer through to the embedding layers, as this would require re-encoding properties and regenerating embeddings at each training step. This architectural constraint also limits the incorporation of advanced techniques, such as attention mechanisms, for optimizing the aggregation of node embeddings into graph-level representations.

One promising method employing end-to-end training for the generation of embeddings of EHR are the BEHRT [33] models, that represent a patient journey as a sequence of encounters, akin to a sequence of words in a sentence, in order to aggregate them using tokenized descriptions of health codes as input. Recent iterations [36] are successfully employing graph learning techniques for embedding generation, following the same hypothesis as this work: that properly encoding the inherent connections in health data improves results. However, the graph representation employed is quite simplistic, lacking in expressiveness compared to a temporal property graph. THis makes it worth investigating if using one would improve the performance by allowing the model to learn from a richer basis.

The difficulty in employing BEHRT models lies in the selection of time representations, as well as training tasks, as there needs to be a discrete set of variables the model can predict for training, in order to generate a loss. Some of these questions have been addressed in a recent analysis [34], which also employed a time2vec layer for encoding temporal information, similarly to this work. For training tasks, using a set of medical codes from existing systems like SNOWMED-CT is preferred, which exist for conditions, procedures, and drugs. However, these codes aren't always universally adapted, making the choice of training data dependent on the potential (geographic) area of deployment.

In conclusion, future work in this area primarily depends on finding appropriate data representations and optimization objectives. As a developing field, few standardized methods exist for generating and evaluating EHR data embeddings. Since assessing embedding utility requires clinical context, approaches allowing for end-to-end training and therefore adaptability to different contests and tasks show promise and warrant further investigation, particularly regarding the efficacy of utilizing graph data.

Bibliography

[1] "Progress on implementing and using electronic health record systems: Developments in OECD countries as of 2021," OECD Health Working Papers 160, Sep. 21, 2023. DOI: 10.1787/4f4ce846-en. [Online]. Available: https://www.oecd-ilibrary.org/social-issues-migration-health/progress-on-implementing-and-using-electronic-health-record-systems_4f4ce846-en (visited on 09/27/2024).

[2] "DB-Engines Ranking per database model category." (), [Online]. Available: https://db-engines.com/en/ranking_categories (visited on 03/30/2024).

[3] R. Angles, "The Property Graph Database Model," presented at the Alberto Mendelzon Workshop on Foundations of Data Management, 2018. [Online]. Available: https://www.semanticscholar.org/paper/The-Property-Graph-Database-Model-Angles/91d6e8ba5dd90b02fe3bd870b19da13a6167af53 (visited on 09/06/2024).

[4] C. Rost, A. Thor, and E. Rahm, "Analyzing Temporal Graphs with Gradoop," *Datenbank-Spektrum*, vol. 19, no. 3, pp. 199–208, Nov. 1, 2019, ISSN: 1610-1995. DOI: 10.1007/s13222-019-00325-8. [Online]. Available: https://doi.org/10.1007/s13222-019-00325-8 (visited on 09/28/2024).

[5] M. Beladev, L. Rokach, G. Katz, I. Guy, and K. Radinsky, "tdGraphEmbed: Temporal Dynamic Graph-Level Embedding," in *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, ser. CIKM '20, New York, NY, USA: Association for Computing Machinery, Oct. 19, 2020, pp. 55–64, ISBN: 978-1-4503-6859-9. DOI: 10.1145/3340531.3411953. [Online]. Available: https://doi.org/10.1145/3340531.3411953 (visited on 03/29/2024).

[6] L. Wang, C. Huang, W. Ma, X. Cao, and S. Vosoughi, "Graph-Level Embedding for Time-Evolving Graphs," in *Companion Proceedings of the ACM Web Conference 2023*, Apr. 30, 2023, pp. 5–8. DOI: 10.1145/3543873.3587299. arXiv: 2306.01012 [cs]. [Online]. Available: http://arxiv.org/abs/2306.01012 (visited on 05/07/2024).

[7] Z. Yang, G. Zhang, J. Wu, J. Yang, Q. Z. Sheng, S. Xue, C. Zhou, C. Aggarwal, H. Peng, W. Hu, E. Hancock, and P. Liò, "State of the Art and Potentialities of Graph-level Learning," *ACM Comput. Surv.*, Sep. 12, 2024, ISSN: 0360-0300. DOI: 10.1145/3695863. [Online]. Available: https://dl.acm.org/doi/10.1145/3695863 (visited on 09/27/2024).

[8] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. "Attention Is All You Need." arXiv: 1706.03762 [cs]. (Aug. 1, 2023), [Online]. Available: http://arxiv.org/abs/1706.03762 (visited on 09/29/2024), pre-published.

[9] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, J. Burstein, C. Doran, and T. Solorio, Eds., Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186. DOI: 10.18653/v1/N19-1423. [Online]. Available: https://aclanthology.org/N19-1423 (visited on 09/27/2024).

[10] N. Reimers and I. Gurevych. "Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks." arXiv: 1908.10084 [cs]. (Aug. 27, 2019), [Online]. Available: http://arxiv.org/abs/1908.10084 (visited on 09/29/2024), pre-published.

[11] S. M. Kazemi, R. Goel, S. Eghbali, J. Ramanan, J. Sahota, S. Thakur, S. Wu, C. Smyth, P. Poupart, and M. Brubaker. "Time2Vec: Learning a Vector Representation of Time." arXiv: 1907.05321 [cs]. (Jul. 11, 2019), [Online]. Available: http://arxiv.org/abs/1907.05321 (visited on 09/11/2024), pre-published.

[12] L. McInnes, J. Healy, and J. Melville. "UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction." arXiv: 1802.03426 [cs, stat]. (Sep. 17, 2020), [Online]. Available: http://arxiv.org/abs/1802.03426 (visited on 08/09/2024), pre-published.

[13] L. van der Maaten and G. Hinton, "Visualizing Data using t-SNE," *Journal of Machine Learning Research*, vol. 9, no. 86, pp. 2579–2605, 2008, ISSN: 1533-7928. [Online]. Available: http://jmlr.org/papers/v9/vandermaaten08a.html (visited on 10/22/2024).

[14] C. Malzer and M. Baum, "A Hybrid Approach To Hierarchical Density-based Cluster Selection," in *2020 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI)*, Sep. 14, 2020, pp. 223–228. DOI: 10.1109/MFI49285.2020.9235263. arXiv: 1911.02282 [cs]. [Online]. Available: http://arxiv.org/abs/1911.02282 (visited on 08/12/2024).

[15] D. Moulavi, P. A. Jaskowiak, R. J. G. B. Campello, A. Zimek, and J. Sander, "Density-Based Clustering Validation," in *Proceedings of the 2014 SIAM International Conference on Data Mining*, Society for Industrial and Applied Mathematics, Apr. 28, 2014, pp. 839–847, ISBN: 978-1-61197-344-0. DOI: 10.1137/1.9781611973440.96. [Online]. Available: https://epubs.siam.org/doi/10.1137/1.9781611973440.96 (visited on 10/15/2024).

[16] "Neo4j Graph Database & Analytics – The Leader in Graph Databases," Graph Database & Analytics. (), [Online]. Available: https://neo4j.com/ (visited on 09/28/2024).

[17] "Why Graph Databases?" Graph Database & Analytics. (), [Online]. Available: https://neo4j.com/why-graph-databases/ (visited on 03/30/2024).

[18] "Introduction - Cypher Manual," Neo4j Graph Data Platform. (), [Online]. Available: https://neo4j.com/docs/cypher-manual/5/introduction/ (visited on 10/23/2024).

[19] "Procedures & Functions - APOC Documentation," Neo4j Graph Data Platform. (), [Online]. Available: https://neo4j.com/docs/apoc/5/overview/ (visited on 10/23/2024).

[20] "Introduction - Neo4j Graph Data Science," Neo4j Graph Data Platform. (), [Online]. Available: https://neo4j.com/docs/graph-data-science/2.6/introduction/ (visited on 03/30/2024).

[21] "Node embeddings - Neo4j Graph Data Science," Neo4j Graph Data Platform. (), [Online]. Available: https://neo4j.com/docs/graph-data-science/2.6/machine-learning/node-embeddings/ (visited on 03/30/2024).

[22] H. Chen, S. F. Sultan, Y. Tian, M. Chen, and S. Skiena. "Fast and Accurate Network Embeddings via Very Sparse Random Projection." arXiv: 1908.11512 [cs]. (Aug. 29, 2019), [Online]. Available: http://arxiv.org/abs/1908.11512 (visited on 08/02/2024), pre-published.

[23] B. Perozzi, R. Al-Rfou, and S. Skiena, "DeepWalk: Online learning of social representations," in *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '14, New York, NY, USA: Association for Computing Machinery, Aug. 24, 2014, pp. 701–710, ISBN: 978-1-4503-2956-9. DOI: 10.1145/2623330.2623732. [Online]. Available: https://doi.org/10.1145/2623330.2623732 (visited on 10/23/2024).

[24] A. Grover and J. Leskovec, "Node2vec: Scalable Feature Learning for Networks," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16, New York, NY, USA: Association for Computing Machinery, Aug. 13, 2016, pp. 855–864, ISBN: 978-1-4503-4232-2. DOI: 10.1145/2939672.2939754. [Online]. Available: https://doi.org/10.1145/2939672.2939754 (visited on 10/23/2024).

[25] P. Li, T. J. Hastie, and K. W. Church, "Very sparse random projections," in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '06, New York, NY, USA: Association for Computing Machinery, Aug. 20, 2006, pp. 287–296, ISBN: 978-1-59593-339-3. DOI: 10.1145/1150402.1150436. [Online]. Available: https://dl.acm.org/doi/10.1145/1150402.1150436 (visited on 10/23/2024).

[26] A. Narayanan, M. Chandramohan, R. Venkatesan, L. Chen, Y. Liu, and S. Jaiswal. "Graph2vec: Learning Distributed Representations of Graphs." arXiv: 1707.05005 [cs]. (Jul. 17, 2017), [Online]. Available: http://arxiv.org/abs/1707.05005 (visited on 05/07/2024), pre-published.

[27] Q. V. Le and T. Mikolov. "Distributed Representations of Sentences and Documents." arXiv: 1405.4053 [cs]. (May 22, 2014), [Online]. Available: http://arxiv.org/abs/1405.4053 (visited on 05/07/2024), pre-published.

[28] Y. Bai, H. Ding, Y. Qiao, A. Marinovic, K. Gu, T. Chen, Y. Sun, and W. Wang. "Unsupervised Inductive Graph-Level Representation Learning via Graph-Graph Proximity." arXiv: 1904.01098 [cs, stat]. (Jun. 2, 2019), [Online]. Available: http://arxiv.org/abs/1904.01098 (visited on 05/07/2024), pre-published.

[29] K. Xu, W. Hu, J. Leskovec, and S. Jegelka. "How Powerful are Graph Neural Networks?" arXiv: 1810.00826 [cs, stat]. (Feb. 22, 2019), [Online]. Available: http://arxiv.org/abs/1810.00826 (visited on 05/07/2024), pre-published.

[30] A. Tsitsulin, D. Mottin, P. Karras, A. Bronstein, and E. Müller, "NetLSD: Hearing the Shape of a Graph," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, Jul. 19, 2018, pp. 2347–2356. DOI: 10.1145/3219819.3219991. arXiv: 1805.10712 [cs]. [Online]. Available: http://arxiv.org/abs/1805.10712 (visited on 05/07/2024).

[31] B. Rozemberczki and R. Sarkar, "Characteristic Functions on Graphs: Birds of a Feather, from Statistical Descriptors to Parametric Models," in *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, ser. CIKM '20, New York, NY, USA: Association for Computing Machinery, Oct. 19, 2020, pp. 1325–1334, ISBN: 978-1-4503-6859-9. DOI: 10.1145/3340531.3411866. [Online]. Available: https://dl.acm.org/doi/10.1145/3340531.3411866 (visited on 09/27/2024).

[32] M. Beladev, G. Katz, L. Rokach, U. Singer, and K. Radinsky, "GraphERT– Transformers-based Temporal Dynamic Graph Embedding," in *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management*, Birmingham United Kingdom: ACM, Oct. 21, 2023, pp. 68–77, ISBN: 9798400701245. DOI: 10.1145/3583780.3614899. [Online]. Available: https://dl.acm.org/doi/10.1145/3583780.3614899 (visited on 05/07/2024).

[33] Y. Li, S. Rao, J. R. A. Solares, A. Hassaine, R. Ramakrishnan, D. Canoy, Y. Zhu, K. Rahimi, and G. Salimi-Khorshidi, "BEHRT: Transformer for Electronic Health Records," *Scientific Reports*, vol. 10, no. 1, p. 7155, Apr. 28, 2020, ISSN: 2045-2322. DOI: 10.1038/s41598-020-62922-y. [Online]. Available: https://www.nature.com/articles/s41598-020-62922-y (visited on 09/24/2024).

[34] M. Odgaard, K. V. Klein, S. M. Thysen, E. Jimenez-Solem, M. Sillesen, and M. Nielsen. "CORE-BEHRT: A Carefully Optimized and Rigorously Evaluated BEHRT." arXiv: 2404.15201 [cs]. (May 22, 2024), [Online]. Available: http://arxiv.org/abs/2404.15201 (visited on 09/16/2024), pre-published.

[35] C. Pang, X. Jiang, K. S. Kalluri, M. Spotnitz, R. Chen, A. Perotte, and K. Natarajan, "CEHR-BERT: Incorporating temporal information from structured EHR data to improve prediction tasks," in *Proceedings of Machine Learning for Health*, PMLR, Nov. 28, 2021, pp. 239–260. [Online]. Available: https://proceedings.mlr.press/v158/pang21a.html (visited on 09/24/2024).

[36] R. Poulain and R. Beheshti, "Graph Transformers on EHRs: Better Representation Improves Downstream Performance," presented at the The Twelfth International Conference on Learning Representations, Oct. 13, 2023. [Online]. Available: https://openreview.net/forum?id=pe0Vdv7rsL (visited on 09/16/2024).

[37] "Synthea." (), [Online]. Available: https://synthetichealth.github.io/synthea/ (visited on 09/27/2024).

[38] J. Walonoski, M. Kramer, J. Nichols, A. Quina, C. Moesel, D. Hall, C. Duffett, K. Dube, T. Gallagher, and S. McLachlan, "Synthea: An approach, method, and software mechanism for generating synthetic patients and the synthetic electronic health care record," *Journal of the American Medical Informatics Association*, vol. 25, no. 3, pp. 230–238, Mar. 1, 2018, ISSN: 1527-974X. DOI: 10.1093/jamia/ocx079. [Online]. Available: https://doi.org/10.1093/jamia/ocx079 (visited on 09/27/2024).

[39] "Generic Module Framework," GitHub. (), [Online]. Available: https://github.com/synthetichealth/synthea/wiki/Generic-Module-Framework (visited on 10/26/2024).

[40] *Neo4jSolutions/patient-journey-model*, Neo4j Solutions, Jun. 23, 2024. [Online]. Available: https://github.com/Neo4jSolutions/patient-journey-model (visited on 10/26/2024).

[41] *Neo4j-field/pyingest*, neo4j-field, Oct. 18, 2024. [Online]. Available: `https://github.com/neo4j-field/pyingest` (visited on 10/26/2024).

[42] *UKPLab/sentence-transformers*, Ubiquitous Knowledge Processing Lab, Oct. 26, 2024. [Online]. Available: `https://github.com/UKPLab/sentence-transformers` (visited on 10/26/2024).

[43] S. K. Sahu, *Ojus1/Date2Vec*, Sep. 25, 2024. [Online]. Available: `https://github.com/ojus1/Date2Vec` (visited on 10/23/2024).

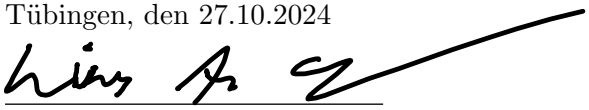
Declaration of Originality

Ich versichere, dass ich die vorliegende Arbeit mit dem Thema:

"*Explorative Graph-Level Embeddings for Temporal Property Graphs*"

selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe, insbesondere sind wörtliche und sinngemäße Zitate als solche gekennzeichnet. Mir ist bekannt, dass Zuwiderhandlung auch nachträglich zur Aberkennung des Abschlusses führen kann. Ich versichere, dass das elektronische Exemplar mit den gedruckten Exemplaren übereinstimmt.

Tübingen, den 27.10.2024



Linus Andreas Schneider