Graph-based Machine Learning for Enhanced Healthcare

Timon Zimmermann
timon.zimmermann@epfl.ch
timon.zimmermann@oracle.com

Thesis submitted for the EPFL degree

Master of Science in Data Science

March 15, 2019

Under the joint supervision of

École polytechnique fédérale de Lausanne
(EPFL)

Distributed Information Systems Laboratory
(LSIR)

Prof. Karl Aberer
karl.aberer@epfl.ch

Oracle Labs Switzerland

Parallel Graph AnalytiX Group (PGX)

Rhicheek Patra
rhicheek.patra@oracle.com

Acknowledgments

First and foremost, I would like to thank all the Oracle Labs team in Zurich for their good mood and friendly welcome. The atmosphere at the office helped keeping the motivation up.

Particularly, I would like to thank my supervisor Rhicheek for his precious experience and insightful advice throughout the whole project.

I would also like to thank Professor Aberer's laboratory (LSIR) and his PhD students for their valuable inputs during the midterm presentation that partially lead to the current solution.

Finally, I am grateful to my family and friends for their unconditional support that helped me get to the end of this Thesis and Master's degree.

Abstract

Healthcare suffers from many manual processes such as correlating measurements, suggesting diagnosis, going through the patient's personal/family history, looking for past similar cases ...

Therefore, we propose an approach to empower physicians and speed-up diagnostic making for patients admitted in Intensive Care Units. This approach shows significant improvement over a single-admission scenario, as we leverage Graph-based ML techniques to convey relational information from previous admissions of other patients.

Therefore, we show how the combination of graphs and advanced ML techniques helps pushing fostering automation in healthcare.

Table of Contents

1
Introduction

This chapter introduces the context of the work being presented here in sections 1.1, to understand the ins and outs of it. Moreover, the problem statement is further developed in sections 1.2 to have a better idea of the problem at hand.
On a related note, the section 1.3 explicitly describes the research question this thesis tries to answer and the different contributions it makes. Finally, the overall structure of the thesis is detailed in the relevant section 1.4.

1.1 Motivation

It is safe to assume that there are multiple manual processes involved in healthcare such as writing summary notes, correlating measurements, suggesting diagnosis, going through the patient's personal/family history, looking for past similar cases... Even with the recent progress made in "Electronic Health Records" the information they provide is not much easier to process than the good old pen and paper. Indeed, even if all these EHR are now easily accessible and normalized, it usually conveys much more information than a doctor can make sense of, at the same time. This EHR data contains information of various kinds, i.e., structured and unstructured, ranging from laboratory measurements to vital signs and even physician notes or radiology imaging.

Recent studies [1] in psychological science show that humans can process at most four interacting variables at the same time, supporting the idea that to computer-assisted healthcare is the future of the domain. Therefore, the primary value that machine learning brings in healthcare is its ability to process huge datasets, beyond the scope of human capability and of different types (e.g. images, texts, ...). Afterwards, to reliably leverage information from that data into clinical insights to aid doctors in providing care, ultimately empowering physicians and speeding-up decision-making, thereby leading to better outcomes, lower costs of healthcare, and eventually improve patient satisfaction.

1.2 Problem Statement

This thesis describes the problem of automatically predicting diagnoses for patients staying in intensive care units. To this end, we employ a restricted database. [1] Comprising of de-identified healthcare data associated with over 40 thousand patients with multiple admissions per patient, making for a bit less than 60 thousand in total.

In our approach, we consider four measurements/events as predictive features per admission: fluids into the patient, fluids out of the patient, lab test results and drugs prescribed by doctors. These events represent the evolving state of a patient during his stay which is then captured by a graph with different nodes for each event, where the topology and linkage of the graph are evolving as new events arrive. On

[1] https://mimic.physionet.org/

top of this, the current admission (for which we want to predict diagnoses) is enriched with information from other admissions, potentially of other patients, by leveraging a knowledge graph that is built from our dataset and external ones.

Our final objective is to feed this evolving graph to a Recurrent Neural Network and predict at every step the probability that of one or many of the top 50 diagnoses will be diagnosed at discharge.

1.3 Research Question & Contributions

Contributions of this thesis are formulated in the following question:

> How to structure our healthcare data, as a graph, and design an appropriate pipeline to leverage as much information as possible from previous admissions of other patients?

To answer this question, this thesis provides the following contributions:

- Propose a way to build an appropriate knowledge graph from admissions data as well as external medical data.
- Suggest how to naturally build an evolving knowledge graph for each admission, to represent the evolution of a patient's state.
- Build a pipeline to leverage this evolving knowledge graph to solve the task at hand: Predicting diagnoses at discharge.
- Analyze the results both quantitatively and qualitatively to further assess the quality of the proposed pipeline.

1.4 Thesis Structure

This thesis starts by a quick introduction in chapter 1 where we overview the motivation that lead to this project and the problem statement, summarized in a research question and different contributions.
The following chapter 2 develops the core concepts that will permit the reader to understand the implementation and difficulties of such a problem.
After that, the chapter 3 introduces the dataset at hand that we will use to apply our proposed framework and pipeline. This chapter goes into details about the dataset, its limitations and scope as well as some statistics detailing it.
Then, we introduce the baseline model in chapter 4 from a conceptual point-of-view and in details. Furthermore, we provide schematic visualizations of this baseline to fully grasp the idea, since the proposed pipeline relies on some parts.
In chapter 5, we go into the details of our proposed framework called *KG-RNN*. The pipeline is detailed step-by-step to allow for an easy re-implementation of each stage.
Moreover, we propose some experiments in chapter 6 to compare the baseline versus KG-RNN. This set of experiments highlights interesting findings on KG-RNN through plots, to analyze both quantitatively and qualitatively the results.
Finally, in the last chapter 7, we conclude this thesis and propose some potential improvements for future work that could lead to an even better architecture for KG-RNN.

2 Background

This chapter will provide the reader with relevant information and background knowledge that are required to understand the proposed architecture. Indeed, the main concepts described hereunder are at the core of the employed methods and rationale behind the different intuitions.

To summarize, we will go through the main benefits of knowledge graphs and why they are gaining traction in the machine learning field. Then, the concept of Recurrent Neural Network (RNN) will be broached, a famous deep learning architecture to induce temporal correlation into a model. Finally, we will explore the concept and body of literature for evolving knowledge graphs and their applications.

2.1 Knowledge Graphs

Knowledge graphs model information through entities, properties of these entities and relationships between them. This kind of modelization of information allows to naturally combine different sources together and ultimately leverage information from data streams initially disparate. By managing to merge together these sources, the goal is to enrich our knowledge on the given entities and eventually leverage this additional linkage structure information with machine learning techniques.

For example, on the figure 2.1, we could imagine that geographical links (e.g. borders, is located in, . . .) come from a different source (e.g. OpenStreeMap) of information than the political links (e.g. is part of, has alternative name, . . .). In this enhanced graph combining information, two entities might be closer to each other (in terms of graph hops) than in any of the individual sources.

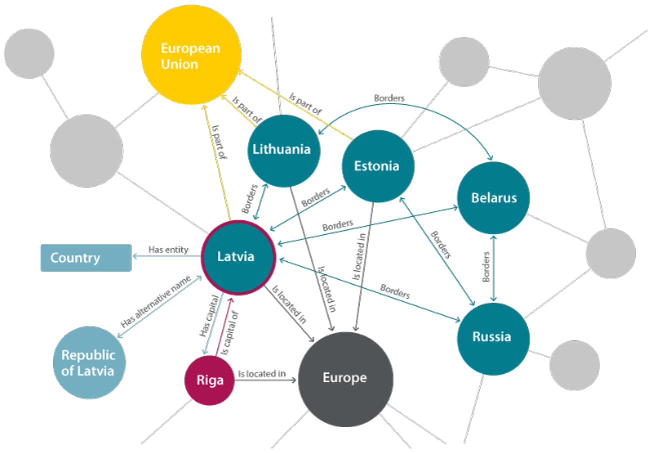


Figure 2.1: An example of a knowledge graph representing how countries are related geographically.

Finally, any algorithmic technique employed downstream could potentially make use of the edge type (e.g. borders, is part of, . . .) in the procedure, by giving more weights to specific edge types, for example. All in all, this allows for a much wider spread of potential techniques and information extraction algorithms or models.

For these reasons, representing and leveraging knowledge as such has a long history in machine learning and computer science as a whole [2, 3]. In the last few years, this concept has gained a lot of traction in the scientific community as well as in the industry [4, 5] for obvious reasons. Indeed, the tremendous progress made in machine learning coupled with this idea of knowledge graph has been applied to many applications such as fraud detection (e.g. for a bank), community identification (e.g. in a social network), recommender systems (e.g. for an online shop). In such systems, it may be difficult for a human to make sense of the different links between entities and how their properties correlate as these links could be multiple hops away from each other.

In these previous examples, the entities, properties and relationships could be the following:

Bank knowledge graph:

Entities and Properties:

- Person(First name, Last name, Address)
- Company(Name, Address)
- Account(Number, Type, Amount)

Relationships:

- Person – *is owner* – Account
- Company – *is owner* – Account
- Person – *work at* – Company
- Account – *wire transfer* – Account

Social network knowledge graph:

Entities and Properties:

- Person(First name, Last name)
- Group(Name)
- Content(Title)

Relationships:

- Person – *friend with* – Person
- Person – *likes* – Content
- Person – *belongs to* – Group

Online shop knowledge graph (figure 2.2):

Entities and Properties:

- Client(First name, Last name)
- Category(Name)
- Product(Name, Price, Properties)

Relationships:

- Client – *purchased* – Product
- Client – *follows* – Category
- Product – *belongs to* – Category
- Product – *shares property* – Product
- Product – *has similar price* – Product

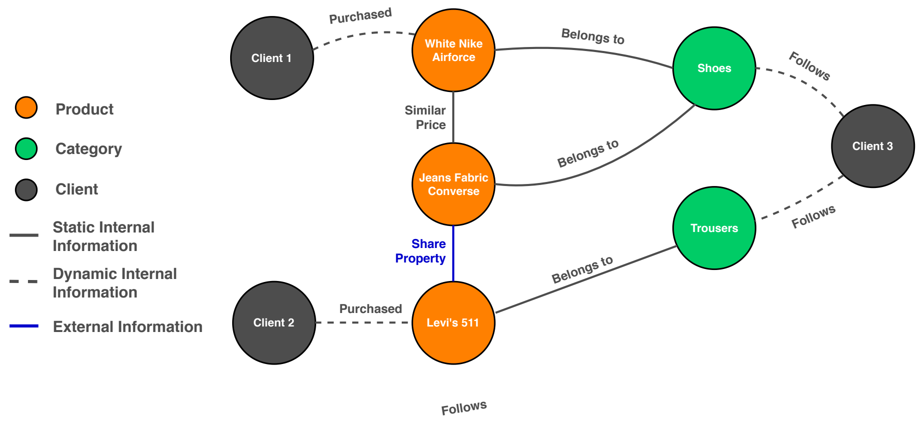


Figure 2.2: An example of evolving knowledge graph for the "Online Shop" case described above. The evolution of the graph is shown through the labels $T = t$ for some edges. Indeed, the clients are buying, and generally performing actions, over time and the structure of the graph is evolving accordingly at different time step. The external information linking these *Converse* shoes (made with jeans fabric) and a *Levi's 511* may come from an external database to automatically map product identifiers to materials and fabrics they are made of.

2.2 Recurrent Neural Network

Recurrent Neural Network consists of a type of architecture for Deep Learning models that allows exploiting temporal dynamic behavior for sequence. This makes them particularly interesting and efficient for tasks that exhibit a temporal pattern, such as speech recognition or market forecasting. On top of that, this kind of architecture allows processing variable-length sequence input, which is not the case for most of the other machine learning (e.g. *linear regression* or *support vector machine*) or deep learning techniques (e.g. *convolutional neural networks*).

Originally, these Recurrent Neural Networks were first heard of in 1986 [6] but the term *recurrent* was not yet used to qualify the progress introduced in the paper. It is only three years later [7] that the term started to appear more actively in the research literature to qualify this type of neural networks.

At core of Recurrent Neural Networks, there are cells that rule how information is blended together from previous time steps and current step, and how much importance is given to these past and present steps. Technically, the RNN handles variable-length inputs by storing a hidden state that is recurrently updated at each time step based on its previous hidden state at time *T-1*, and the current input at time *T*. More formally, given a sequence of inputs $\boldsymbol{x} = (\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_T)$, the hidden state $\boldsymbol{h}_t$ is updated following the equation:

$$\boldsymbol{h}_t = \begin{cases} \mathbf{0}, & t = 0 \\ f(\boldsymbol{h}_{t-1}, \boldsymbol{x}_t; \boldsymbol{\theta}), & \text{otherwise} \end{cases} \tag{2.1}$$

With f refers to a non-linear function depending on some parameters $\boldsymbol{\theta}$. Typically, the equation 2.1 is defined as:

$$\boldsymbol{h}_t = f(U\boldsymbol{x}_t + W\boldsymbol{h}_{t-1} + \boldsymbol{b}) \tag{2.2}$$

Where the parameters $\boldsymbol{\theta}$ are the matrices W, U and the bias term $\boldsymbol{b}$, while the function f is a smooth and bounded function such as sigmoid or a hyperbolic tangent. Finally, the sequence of outputs of the RNN $\boldsymbol{y} = (\boldsymbol{y}_1, \boldsymbol{y}_2, ..., \boldsymbol{y}_T)$ is usually defined as:

$$\boldsymbol{y}_t = g(V\boldsymbol{h}_t + \boldsymbol{c}) \tag{2.3}$$

Where the matrix V and the bias vector $\boldsymbol{c}$ are learned parameters, while the function g is often a *softmax*.

Unfortunately, such networks are suffering from many problems while training to use *back-propagation through time* or ($BPTT$). Indeed, this architecture has difficulties capturing long-term dependencies in the input sequence because the gradient update has a tendency to take abnormally big values (i.e. explode) or small ones (i.e.e vanish, since they are squashed at every time steps by the bounded activation functions *sigmoid* or *tanh*).

To circumvent these two problems, researchers have designed more sophisticated activation functions, called cell in this case. The two principal types of cell are LSTM [8] and GRU [9], the former predating the latter by almost 20 years. A schematic view of these types of cell can be found in 2.3 for a better understanding of the mechanisms ruling the learning dynamic. These are the typical go-to cells for any application requiring a Recurrent Neural Network. However there does not exist a rule of thumb to pick one of the two as this is usually dependent on the application and, more precisely, on the data at hand [10]. Nonetheless, one interesting argument in favor of the GRU cell is that it depends on fewer parameters than LSTM, as they lack an output gate. It makes them particularly interesting when the computational resources are limited or when one wants to iterate over quick experiments.

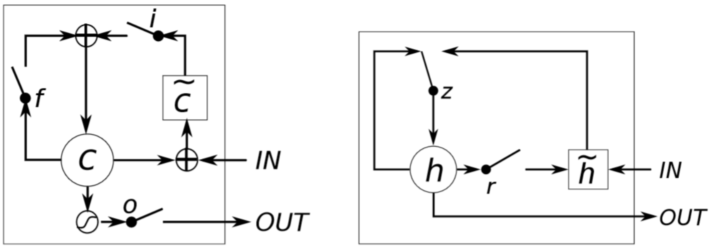


Figure 2.3: The schematic views of an LSTM (*left*) and GRU (*right*) cell. On the left-hand cell, i, o and f mean respectively *input*, *output* and *forget* gates. Finally, on this same cell, c and $\tilde{c}$ represents the old and new contents of the memory cell. For the right-hand cell, r and z are the reset and update gates while h and $\tilde{h}$ are respectively the actual and new (or candidate) activation. Figure from [10].

The equations defining the update dynamics of the hidden state, as well as the other parameters in these cells are the following:

LSTM update equations

$$\begin{aligned}
\boldsymbol{i}_t &= \sigma(U^i\boldsymbol{x}_t + W^i\boldsymbol{h}_{t-1})\\
\boldsymbol{f}_t &= \sigma(U^f\boldsymbol{x}_t + W^f\boldsymbol{h}_{t-1})\\
\boldsymbol{o}_t &= \sigma(U^o\boldsymbol{x}_t + W^o\boldsymbol{h}_{t-1})\\
\tilde{\boldsymbol{C}}_t &= tanh(U^g\boldsymbol{x}_t + W^g\boldsymbol{h}_{t-1})\\
\boldsymbol{C}_t &= \sigma(\boldsymbol{f}_t \odot \boldsymbol{x}_t + \boldsymbol{i}_t \odot \tilde{\boldsymbol{C}}_t)\\
\boldsymbol{h}_t &= tanh(\boldsymbol{C}_t) \odot \boldsymbol{o}_t
\end{aligned}$$

GRU update equations

$$\begin{aligned}
\boldsymbol{z}_t &= \sigma(U^z\boldsymbol{x}_t + W^z\boldsymbol{h}_{t-1})\\
\boldsymbol{r}_t &= \sigma(U^r\boldsymbol{x}_t + W^r\boldsymbol{h}_{t-1})\\
\tilde{\boldsymbol{h}}_t &= tanh(U^h\boldsymbol{x}_t + W^h(\boldsymbol{r}_t \odot \boldsymbol{h}_{t-1}))\\
\boldsymbol{h}_t &= (1-\boldsymbol{z}_t) \odot \boldsymbol{h}_{t-1} + \boldsymbol{z}_t \odot \tilde{\boldsymbol{h}}_t
\end{aligned}$$

Finally, these two cells and all these equations are defining a *causal* structure. That is, the output at time T only depends and capture information from the past $1, ..., T-1$ through the learning dynamics.

However, it may be interesting to have the output at each time step to *depend on the whole sequence*, in applications where it makes sense. Indeed, if one is doing part-of-speech tagging, the tag of a given work not only depends on the preceding words but following ones. On the contrary, if one wants to run the model on an *online* stream of data for example, it would be flawed to train the model with access to future time steps.

However, in many cases it is possible to train such models with information from previous and next states where the relevant architecture is called *Bidirectional RNN*. This kind of model combines an RNN that moves forward and one that moves backward, to finally mix their respective output with a concatenation followed by a fully-connected layer.

To conduct our research, we picked GRU cells in our case for the reasons mentioned above, namely their efficiency that allows to iterate over different architectures and hyper-parameters faster. The use of bidirectional RNN or not will be a hyper-parameter tuned with the other ones.

2.3 Evolving Knowledge Graph

This last core concept tackles the problem of evolving knowledge graph, where the term *evolving* refers to a dynamic component within the knowledge graph. Furthermore, it characterizes changes in the graphs that can be two fold: changes in the structure (e.g. new entities and links, such as a new *company* in the "Bank knowledge graph" example in section 2.1), changes in properties within entities (e.g. the *product price* could evolve over time for the "Online shop knowledge graph").

Ultimately, the goal is to account for such dynamic behavior in the structure of the graph or its properties. Indeed, one could consider these changes as a sequence of graph states, each state representing the graph structure and property values at a given time T. This additional information would benefit many applications and allow for much complex information interaction than a knowledge graph that remains static. However, incorporating temporal information into knowledge graph-based learning is still one of the major difficulties [4]. For this reason, a lot of efforts have been carried around the topic [11–14], but there still remains a lot of challenges such as scalability to large real-world applications and capacity to capture dynamics in skewed graphs [15].

With this in mind, we are seeking an architecture that is able to reason over time by capturing the different dynamics of the knowledge graph at the same time as leveraging its structure and relationships.

3
Healthcare Application & Dataset

This chapter will provide the reader with relevant information about the dataset at hand. Indeed, the idea of our approach, *KG-RNN*, is quite general but we demonstrate its usefulness in a concrete application in this thesis.

In particular, we develop the concept of *Enhanced Healthcare* and apply our technique to the diagnosis of disease at a patient discharge. To sum up the structure of chapter, we will introduce the dataset in section 3.1, its source as well as its limitation and scope. Besides, we will explain the preparation procedure of the dataset in section 3.2, how we are processing it and formatting it to be ready for modeling. Finally, we will describe and present statistics about the dataset in section 3.3 in order for the reader to have a better grasp of the big picture, the distributions and possibilities offered.

All of this bearing in mind our end goal: **Predict the one or multiple diagnoses assigned to a patient at discharge, among the top-50 most popular ones, given the different events occurring for this patient (measurements, medication, . . .).**

3.1 Description

To tackle our task of enhancing and improving healthcare by predicting diagnoses of patients at discharge of their ICU stay, we decided to utilize the MIMIC-III dataset [2]. This dataset includes over 40 thousand patients with multiple admissions per patient, making for a bit less than 60 thousand in total. All the information is de-identified and subject to restricted database, requiring going through a training program before being able to download any file. All in all, the data spans from June 2001 to October 2012.

As a high level, the main component of the dataset is *admissions*; additionally other tables provide information about the different measurements taken throughout a patient time span as well as prescribed drugs. Finally, we are given the diagnoses at discharge, that is both diseases and procedures for the patient.

These diagnoses at discharge are represented in the form of ICD9 (*International Classification of Diseases*) codes, which are maintained by the World Health Organization. The purpose of the system is to create a mapping between conditions and generic categories, with different levels of granularity. Indeed, this classification forms a tree where upper-level branches are including the set of similar diseases with specific variations.

A database of all the ICD codes for version 9 (*ICD9*) can be found at http://www.icd9data.com/ and here are a few examples to get the reader acquainted with the system:

[2] https://mimic.physionet.org/

ICD9 code or range	**Description**
001-139	Infectious and Parasitic Diseases.
010-018	Tuberculosis.
011	Pulmonary tuberculosis.
011.0	Tuberculosis of lung infiltrative.
011.03	Tuberculosis of lung, infiltrative, tubercle bacilli found (in sputum) by microscopy.

In the following tables, we will enumerate and describe the different fields available to our research. We filtered these fields and eventually only describe the ones we are using and that is of interest for our application at hand but many more are available in the *MIMIC-III* dataset.

Table containing all the admissions from the different patients:

Table 3.1: Admissions

Name	**Description**	**Example value**
SUBJECT_ID	Unique identifier for the patient.	23
HADM_ID	Unique identifier for the admission.	124321
ADMITTIME	Time at which the patient was admitted to the hospital.	2157-10-18 19:34:00
DISCHTIME	Time at which the patient was discharged from the hospital.	2157-10-25 14:00:00
DIAGNOSIS	Preliminary, free text diagnosis for the patient on hospital admission.	BRAIN MASS

Table containing the different ICD9 codes diagnosed at discharge of the ICU stay:

Table 3.2: Diagnoses

Name	**Description**	**Example value**
SUBJECT_ID	Unique identifier for the patient.	62641
HADM_ID	Unique identifier for the admission.	154460
ICD9_CODE	Diagnoses assigned to the patient at discharge for a given admission.	3404
SEQ_NUM	Priority number of the given ICD9 code for the patient's admission.	3

Table containing the different laboratory measurements on a patient during his stay at the ICU:

Table 3.3: Laboratory measurements

Name	**Description**	**Example value**
SUBJECT_ID	Unique identifier for the patient.	3
HADM_ID	Unique identifier for the admission.	145834
ITEMID	Unique identifier for the measurement type in the database.	50868 (*Bicarbonate*)
CHARTTIME	Time at which the observation was charted.	2101-10-20 16:40:00
VALUENUM	Value measured for the specific measure *ITEMID*.	17.0
VALUEUOM	Unit of measurement for the measure *ITEMID*.	mEq/L

Table containing the different input fluids administered to a patient during his stay at the ICU:

Table 3.4: Input events from CareVue (Electronic Medical Records system)

Name	**Description**	**Example value**
SUBJECT_ID	Unique identifier for the patient.	24457
HADM_ID	Unique identifier for the admission.	184834
ITEMID	Unique identifier for the fluid type in the database.	30056 (*Po Intake*)
CHARTTIME	Time at which the observation was charted.	2193-09-11 09:00:00
AMOUNT	Amount of the drug or substance administered *ITEMID*.	100.0
AMOUNTUOM	Unit of measurement of the drug or substance administered *ITEMID*.	ml

Table containing the different input fluids administered to a patient during his stay at the ICU:

Table 3.5: Input events from Metavision (Electronic Medical Records system)

Name	**Description**	**Example value**
SUBJECT_ID	Unique identifier for the patient.	27063
HADM_ID	Unique identifier for the admission.	139787
ITEMID	Unique identifier for the fluid type in the database.	225944 (*Sterile Water*)
STARTTIME	Start time of the injection.	2133-02-05 05:34:00
ENDTIME	End time of the injection.	2133-02-05 06:30:00
AMOUNT	Amount of the drug or substance administered *ITEMID*.	100.120008
AMOUNTUOM	Unit of measurement of the drug or substance administered *ITEMID*.	ml
RATE	Rate at which the drug or substance *ITEMID* was administered.	30.036002
RATEUOM	Unit of measurement of the rate at which the drug or substance *ITEMID* was administered.	ml/hour

Table containing the different output fluids measured from a patient during his stay at the ICU:

Table 3.6: Output events from CareVue and Metavision

Name	**Description**	**Example value**
SUBJECT_ID	Unique identifier for the patient.	21219
HADM_ID	Unique identifier for the admission.	177991
ITEMID	Unique identifier for the fluid type in the database.	40055 (*Urine Out Foley*)
CHARTTIME	Time at which the output fluid was charted.	2142-09-08 10:00:00
VALUENUM	Value measured for the fluid *ITEMID*.	200.0
VALUEUOM	Unit of measurement for the fluids *ITEMID*.	ml

Table containing the different medications prescribed to a patient during his stay at the ICU:

Table 3.7: Prescriptions

Name	**Description**	**Example value**
SUBJECT_ID	Unique identifier for the patient.	6
HADM_ID	Unique identifier for the admission.	107064
STARTDATE	Start time of the medication's prescription.	2175-06-11 00:00:00
ENDDATE	End time of the medication's prescription.	2175-06-13 00:00:00
DRUG	Identify the type of drug prescribed in the database.	Warfarin
DRUG_VAL_RX	Amount of the medication prescribed *ITEMID*.	5
DRUG_UNIT_RX	Rate at which the medication *ITEMID* was prescribed.	mg

Limitation & Scope MIMIC-III is a dataset that is de-identified, as explained briefly in the previous session. That is, we do not have access to many personally identifiable information (*PII*) for the patient for obvious confidentiality reasons.

For example, all dates in the dataset have been randomly shifted and are therefore not possible to correlate between patients. This is to protect patient confidentiality and times only make sense to interpret at patient scope. This limitation has implications for any downstream application, such as the impossibility to account for medical trends or discoveries in the course of time.

Indeed, some new drugs may be discovered or innovative procedures to cope with a particular disease. On top of that, the differential diagnosis process has probably evolved over the years covered by the *MIMIC-III* (2001-2012) and thus particular events or measurements could lead to additional or new diseases diagnosed at discharge.

3.2 Preparation

As any real dataset, MIMIC-III is a bit messy and not perfectly structured and normalized. In the following section, we will introduce the various pre-processing and cleaning applied to the dataset to finally talk about dataset formation.

The purpose of this section is to ensure a good understanding of the mechanics of the pipeline as well as full reproducibility of the experiments in chapter 6.

3.2.1 Pre-processing & Cleaning

This subsection will be divided in paragraphs, one for each table described in section 3.1. Each of these paragraph will contain detailed information regarding the processing and cleaning of this specific data.

Admissions For this table, we had to simply remove NaN entries that were present.

Diagnoses For this table, we removed NaN entries that were present but also merged procedures and diseases diagnoses. Finally, for readability, we translated ICD9 codes to their classical representation (from MIMIC-III: *42019* to *420.19*).

Laboratory measurements For this table, we removed NaN entries that were present, we cleaned the units of measurement to lower-case, removed leading and trailing spaces. Finally, we normalized the units of measurements and their values for the same item.

For example, *Testosterone level* is charted as *ng/dl* and *pg/dl*, so we decided to convert all the former to the latter and multiply their respective value by *10*.

Input events from CareVue For this table, we removed NaN entries that were present, we cleaned the units of measurement to lower-case, removed leading and trailing spaces. Finally, we normalized the units of measurements and their values for the same item. For example, there exists *gtt* and *drop* for the same item, or *g* and *mg*.

Input events from Metavision For this table, we removed NaN entries that were present, we cleaned the units of measurement to lower-case, removed leading and trailing spaces. Finally, we normalized the units of measurements and their values for the same item. For example, there exists *gtt* and *drop* for the same item, or *g* and *mg*.

On top of that, input events in *Metavision* are charted in the form of continuous events. That is, a time range is defined as well as a rate and a total amount. Hence, to normalize the data according to *CareVue*, we converted these ranges to a series of one-time events. To do so, we simply divided the total amount of input fluid provided by the size, we then take the rate at which it is administrated (e.g. 1 ml per hour), and create a series of one-time events of this amount at this rate.

Output events from CareVue and Metavision For this table, we removed NaN entries that were present, we cleaned the units of measurement to lower-case, removed leading and trailing spaces. Finally, we normalized the units of measurements and their values for the same item. For example, there exists *gtt* and *drop* for the same item, or *g* and *mg*.

Prescriptions For this table, we removed NaN entries that were present, we cleaned the units of measurement to lower-case, removed leading and trailing spaces. Finally, we normalized the units of measurements and their values for the same item. For example, there exists *gtt* and *drop* for the same item, or *g* and *mg*.

Finally, in a similar manner as *input events from Metavision*, prescriptions are charted as daily ratio of drugs that should be taken between two dates. Therefore, we converted these ranges to one-time events of daily doses taken at lunch.

After cleaning and processing the different tables, each admission is split in chunks of τ hours. From there, each chunk contains the events that were charted during that period (laboratory measurements, input events, output events, prescriptions), as a group.

These groups/chunks can be seen the patient state at time T, this helps deal with all the missing values present in the dataset. Indeed, for the task at hand, we are not interested in the exact value of each patient's vital, but rather by the evolution of his state throughout the admission. To this end, discarding missing values and grouping within a given period makes sense rather than trying to impute missing values, dynamically while training [16] or upstream.

3.2.2 Formation

Dataset formation is not a subtle task as this will dictate how downstream models take the data in. This step of the pipeline starts from the previous stage where we cleaned and pre-processed the different tables.

To get started, we load the client and pre-processed data from the previous section. From there, we start by splitting the dataset in training, validation and testing sets. To do so, we consider patients instead of admissions as we don't want the same patient to be both in the training and testing sets since a few of them have multiple admissions. Surely, training on one admission and testing on another from the same patient would leak information between training and testing set and thus introduce bias in performance metrics.

When the different sets are created, we start by padding or truncating the admissions, depending on whether they are respectively shorter or longer than a pre-defined length. This length N is defined with respect to the number of chunks/groups representing the patient state, so the total admission length considered is computed as $N\tau$.

The different admissions are thus padded with zeros, on the left-hand side (so back in time, temporally), which means that for an admission consisting of 3 chunks, or states, $\boldsymbol{a_i} = [\boldsymbol{c_1}, \boldsymbol{c_2}, \boldsymbol{c_3}]$, it would be transformed as follows:

$$\boldsymbol{a_i} = [\boldsymbol{c_1}, \boldsymbol{c_2}, \boldsymbol{c_3}] \quad \xrightarrow{\text{pad}} \quad \boldsymbol{a_i^*} = [\boldsymbol{0}, \boldsymbol{0}, \dots, \boldsymbol{c_1}, \boldsymbol{c_2}, \boldsymbol{c_3}], \text{ where } |\boldsymbol{a_i^*}| = N$$

While for a longer admission $\boldsymbol{a_i} = [\boldsymbol{c_1}, \boldsymbol{c_2}, \boldsymbol{c_3}, \dots, \boldsymbol{c_{178}}]$, we would have:

$$\boldsymbol{a_i} = [\boldsymbol{c_1}, \boldsymbol{c_2}, \boldsymbol{c_3}, \dots, \boldsymbol{c_{178}}] \quad \xrightarrow{\text{truncate}} \quad \boldsymbol{a_i^*} = [\boldsymbol{c_1}, \boldsymbol{c_2}, \boldsymbol{c_3}, \dots, \boldsymbol{c_{150}}]$$

On top of this, each chunk is padded and truncated from its events when more than K of them happen within a specific of c_i duration τ. That is, we limit the number of events per c_i in the equations above. Therefore, the events within each chunk are also zero-padded if the number of events is lower than K, and randomly sampled if it exceeds K.

Consequently, we have fixed number of chunks (*admission length*) and fixed number of events per chunk. As a result, we are able to batch the admissions and leverage computational efficiency to reduce the training time, which is the main rational between these truncating and padding procedures. Finally, the packed admissions are cast to *float32* from *float64*, in order to reduce memory footprint at the cost of a very slight decrease in terms of value precision.

3.2.3 Schematic Visualizations

The goal of this subsection is to show with a schema the results induced by the previous sections *Pre-processing & Cleaning* and *Formation*. It is important to note that in these visualizations, input events are only shown in the form of the Metavision system (*input events MV*), as ranges in the input row (*green*).

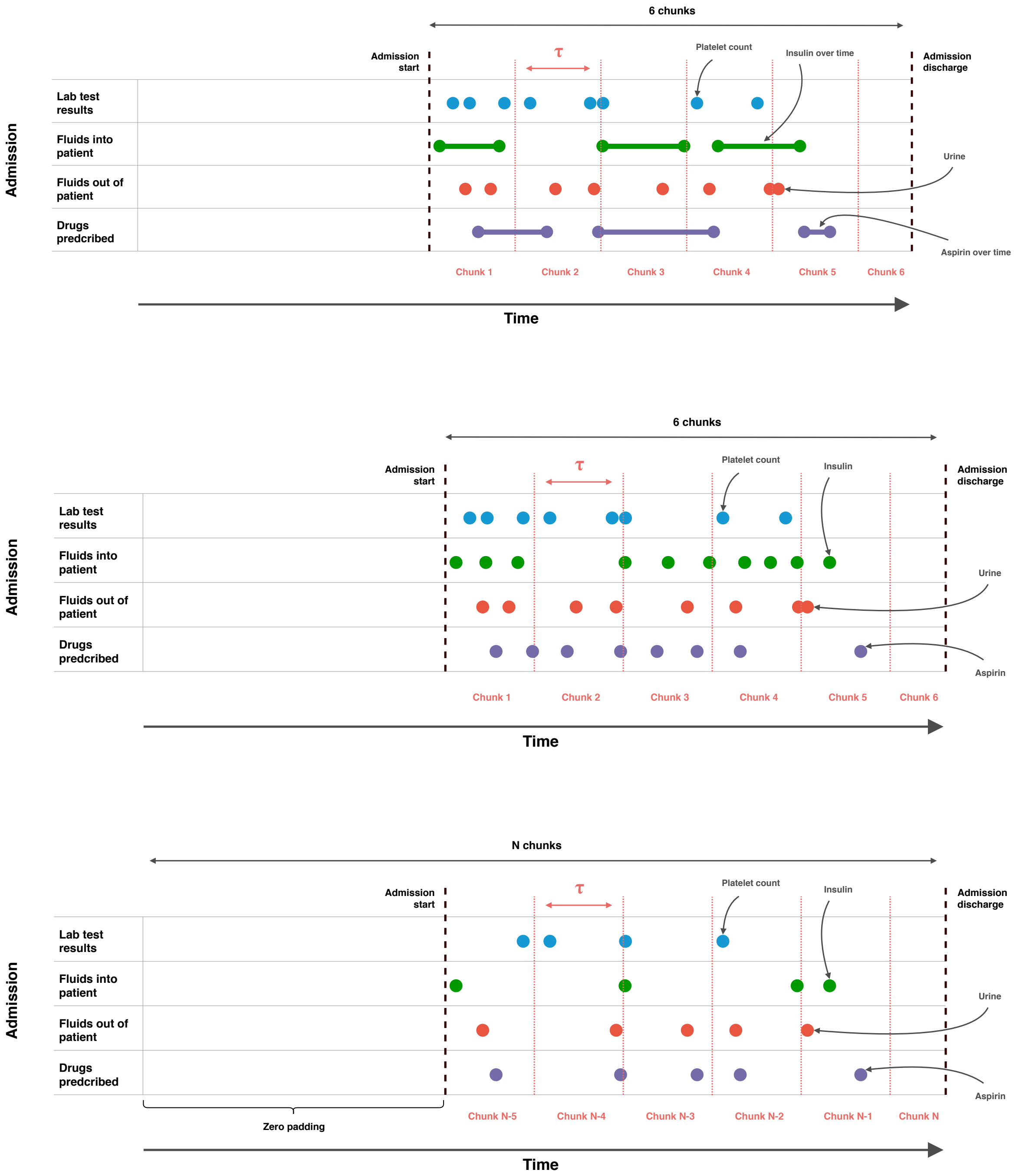


Figure 3.1: **Top row**: The visualization of a single admission, consisting of 6 chunks of duration τ, each with their respective one-time or range events. **Middle row**: The pre-processed and cleaned admission, range events have been converted to one-time events. **Bottom row**: The final version of the admission that will be fed to the downstream model. The admission is padded and events are sampled with $K = 1$, thus we only have 1 event per type and per chunk of duration τ. Chunks with less than $K = 1$ events are also *zero padded* (not shown) to match the K number of events per type and per chunk.

3.3 Statistics

This section describes the different statistics related to the data we have discussed until now, to give some quantitative metrics. In effect, the reader will be able to grasp the possibilities and distributions.

Table 3.8: Dataset size and unique values for each table

Table	Number of events	Number of types
Admissions	58’976	-
Laboratory measurements	19’306’086	368
Input events CV	11’651’110	2’836
Input events MV	80’393’617	165
Output events	4’217’736	1’113
Prescriptions	14’728’948	4’465
Diagnoses	891’142	9’016

Here are some statistics regarding each state/chunk (with $\tau = 3$ hours) and the different events happening throughout the admission timeline:

Number of chunks

Statistic	Value
Minimum	1
Maximum	2’358
Average	81.58
Median	52
95th percentile	247
99th percentile	511

Laboratory measurements per chunk

Statistic	Value
Minimum	0
Maximum	153
Average	3.76
Median	0
95th percentile	25
99th percentile	37

Input CV events per chunk

Statistic	Value
Minimum	1
Maximum	189
Average	2.46
Median	0
95th percentile	15
99th percentile	28

Input MV events per chunk

Statistic	Value
Minimum	0
Maximum	36’400
Average	15.95
Median	0
95th percentile	19
99th percentile	464

Output events per chunk

Statistic	Value
Minimum	0
Maximum	41
Average	0.89
Median	0
95th percentile	4
99th percentile	7

Prescriptions per chunk

Statistic	Value
Minimum	0
Maximum	7’321
Average	1.75
Median	0
95th percentile	9
99th percentile	40

These different statistics helped us define the hyper-parameters related to the padding or truncation of admissions (N) as well as the number of events per chunk (K).

Finally, we show some plots and histograms to enlighten the reader regarding the task at hand, which is to predict the diagnosed top 50 most frequent ICD9 codes at discharge in a multi-class multi-classification fashion:

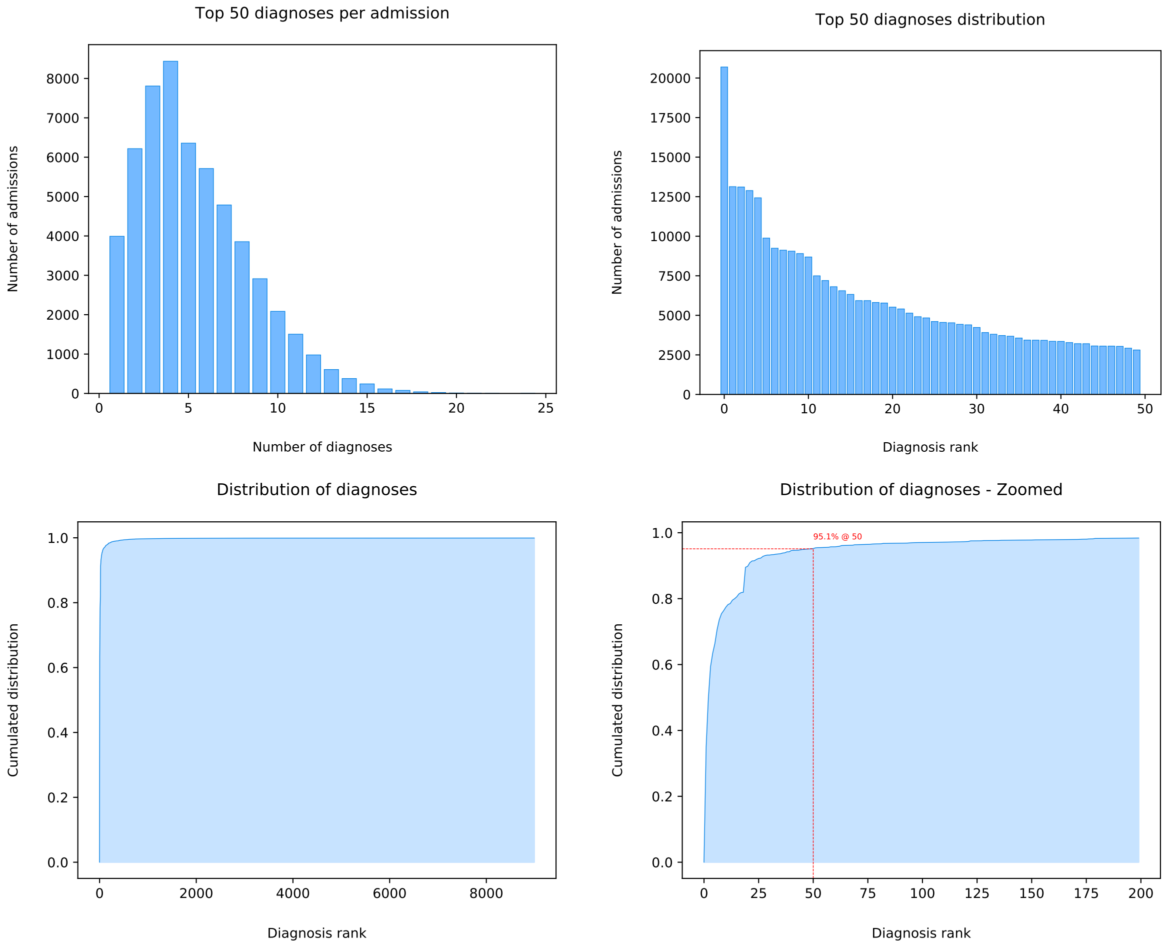


Figure 3.2: **Top-left**: This histogram represents the number of diagnoses per admission at discharge for the top 50 ICD 9 codes, which is interesting to bear in mind in our setup of multi-class multi-classification. **Top right**: This histogram represents the number of occurrences among the top 50 diagnoses, this is an indicator of the class imbalance or not. **Bottom-left**: The cumulative distribution of diagnoses over admissions, we can see that a few diagnoses span more than 90% of the admissions very rapidly. **Bottom right**: The zoomed cumulative distribution of diagnoses over admissions, focusing on the top 200 most frequent diagnoses. We can see that we cover 95.1% of admissions with the top 50 most frequent ICD9 codes.

4 Baseline

This chapter introduces the baseline architecture, which processes an evolving graph. First off, we will go through an overview of the task and a rather "textual" description of the model.

Then, the baseline model will be described in a more precise way as well as a schematic visualization of the operations that are performed.

4.1 Overview

The baseline architecture deals with the problem of processing a single graph, evolving over time. This graph that will be fed to the downstream model is represented by a series of states.

In our case the input entities of interest in our graph are admissions, and the simplest way to predict the diagnoses at discharge of a given input admission is to only consider its events over time as our evolving graph and not directly considering the information from other admission entities. That is, in this baseline approach, we do not consider a knowledge graph linking admissions together within a complex structure, but rather admissions as individual evolving graphs consisted of chunk and event entities.

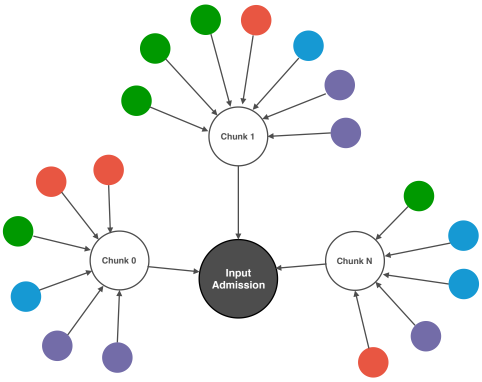


Figure 4.1: Example admission (*without* zero padding), where chunk entities have a natural ordering and event entities do not.

Such a graph can be represented in a compact form as shown in the figure 4.1, the colored entities refer to particular events during the admission timeline, associated to a given chunk depending on the chart time. This admission has a set of diagnoses at discharge and we are looking for an architecture that would enable us to capture the dynamic behavior as well as make sense of the different events.

Therefore, it is natural that we base our research and baseline architecture on a Recurrent Neural Network to deal with the different chunks or states of the graph, this type of architecture allowing exploiting temporal dynamic behavior for sequence. Consequently, we need to input an embedded representation of the graph state at every time step T in our Recurrent Neural Network. That is, we need to *map* our graph state into a finite D-dimensional vector (where D is a hyper-parameter to be chosen appropriately) that will enable us to train the model end-to-end. Indeed, as opposed to older techniques that embed the graph in a separate task, making the back-propagation truncated between the RNN and the graph embedding algorithm, we want our architecture to be fully differentiable.

4.2 Machine Learning Model

Unlike performing machine learning over classical lattices (e.g. images), the neighboring entities or nodes (i.e. *events*) to our graph states do not have a natural ordering. For this reason, owing to GraphSAGE [17], we base our “graph state embedder“ on aggregator functions for each entity type. These functions must operate over an unordered set of vectors and thus be invariant to permutations of its elements.

However, before discussing the aggregator functions in detail, the different events as well as their respective value should be “translated“ in vector forms. As an illustration, let’s consider the following event from the **Prescriptions table**:

$$e^{prescription} = (\text{Warfarin}, 5)$$

This tuple represents an entity of the type *prescription event*, where the patient took 5 units of Warfarin. From there, we create a **prescription embedding matrix** that maps from “Warfarin“ to a n-dimensional vector, where n is a hyper-parameter to be tuned carefully. Finally, this **prescription embedding matrix**, $M \in \mathbb{R}^{P \times n}$ where P represents the number of prescriptions (4’465 in our case), is initialized with zero-vectors.

At the beginning of the training procedure, each prescription event is mapped to its corresponding n-dimensional vector and concatenated to its associated value:

$$\tilde{e}^{prescription} = [\boldsymbol{x}, 5], \text{ where } \boldsymbol{x} \in \mathbb{R}^n$$

At the end of this “translation“ procedure, each prescription event entity is a $(n+1)$-dimensional vector that can be further processed. In a similar fashion, we create an embedding matrix for each event types, resulting in 5 different embedding matrices initialized with zero-vectors that will allow to “translate“ every event entities in the evolving graph.

Hence, a chunk i can be represented as a tensor of shape $5 \times K \times (n+1)$, resulting from the concatenation of the different event types. Thus, an admission can be seen as a series of chunks, leading to a tensor of shape $N \times 5 \times K \times (n+1)$ for each admission.

As mentioned previously, these aggregators should be invariant to permutations of the input, trainable and also have a high representational capacity. To this end, each *aggregator* will first consist of a fully-connected network and followed by the aggregation function, we propose three candidates:

Max aggregator The first nominee consists in element-wise max-pooling operation along the K dimension. Namely, we have the following end-to-end aggregator:

$$\begin{aligned} AGG_m^{max} &= \max_{i \in \{0 \dots K\}} \sigma(\boldsymbol{W}\tilde{\boldsymbol{e}}_i^m + \boldsymbol{b}), \text{ where } \tilde{\boldsymbol{e}}_i^m \in \mathbb{R}^{n+1} \\ &= \max_{i \in \{0 \dots K\}} \boldsymbol{h}_i^m, \text{ where } \boldsymbol{h}_i^m \in \mathbb{R}^a \end{aligned} \tag{4.1}$$

Here max denotes the element-wise max operator along the K dimension, σ is a nonlinear activation function (e.g. ReLU) and a is an arbitrary dimensionality that has to be manually tuned. Besides, m represents the event type among laboratory measurements, input events (CV or MV), output events and prescriptions, while $\boldsymbol{W} \in \mathbb{R}^{a\times(n+1)}$ and $\boldsymbol{b} \in \mathbb{R}^{a}$ are parameters optimized during the back-propagation training.

All in all, for each event type m, the equation 4.3 maps each embeddings to a a-dimensional hidden representation $\boldsymbol{h}_i^m$ and then squashes the events of a similar type with the max operator.

Mean aggregator This aggregator is very similar to the *max aggregator* but performs a mean-pooling operation along the K dimension:

$$\begin{aligned} AGG_m^{mean} &= \underset{i\in\{0\ldots K\}}{\text{mean}} \sigma(\boldsymbol{W}\tilde{\boldsymbol{e}}_i^m + \boldsymbol{b}), \text{ where } \tilde{\boldsymbol{e}}_i^m \in \mathbb{R}^{n+1} \\ &= \underset{i\in\{0\ldots K\}}{\text{mean}} \boldsymbol{h}_i^m, \text{ where } \boldsymbol{h}_i^m \in \mathbb{R}^{a} \end{aligned} \tag{4.2}$$

Sum aggregator Our final candidate looks like the two previous ones, by replacing the element-wise operator with a sum operation:

$$\begin{aligned} AGG_m^{sum} &= \underset{i\in\{0\ldots K\}}{\text{sum}} \sigma(\boldsymbol{W}\tilde{\boldsymbol{e}}_i^m + \boldsymbol{b}), \text{ where } \tilde{\boldsymbol{e}}_i^m \in \mathbb{R}^{n+1} \\ &= \underset{i\in\{0\ldots K\}}{\text{sum}} \boldsymbol{h}_i^m, \text{ where } \boldsymbol{h}_i^m \in \mathbb{R}^{a} \end{aligned} \tag{4.3}$$

A visual explanation is available on the figure 4.2. In essence, it is important to note that all of these aggregators are both symmetric and trainable. Once stacked, the output of these aggregators is a tensor of shape $N \times 5 \times a$ that we further reshape and concatenate to $N \times 5a$ in order to be fed to our downstream Recurrent Neural Network. Thus, referring to the previous section, each $5a = D$ vector is our representation of the graph state at the time step T.

This Recurrent Neural Network outputs a vector of size $\tilde{\boldsymbol{h}} \in \mathbb{R}^b$ that is represented the embedding of our **evolving entity**. In our practical use-case to predict the ICD9 codes at discharge, we simply feed this vector into a fully-connected layer mapping to our *50* classes.

More generally, this embedding algorithm followed by a Recurrent Neural Network is applicable to any arbitrary entity and we call it an **Evolving Entity Encoder**. We present the baseline model applied to our "Healthcare Dataset" but it generalizes easily to different graphs and structures, since it is independent of the number of neighbors.

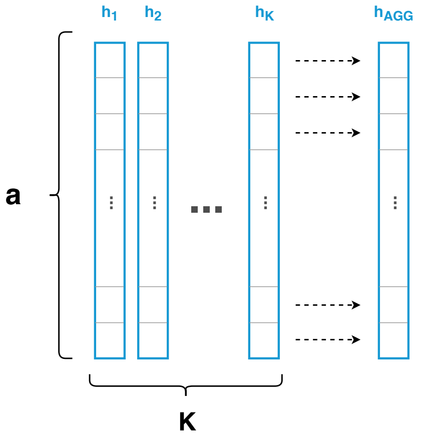


Figure 4.2: An example of an aggregator on a certain event type (*blue*), the dashed arrow represent the type of aggregating function (i.e. sum, mean, max) applied element-wise (row-by-row on the figure). The output is a vector $\boldsymbol{h}_{AGG} \in \mathbb{R}^a$.

4.3 Schematic Visualization

To better understand how the baseline architecture works and generalizes to arbitrary graphs, we propose the following visualizations to the reader:

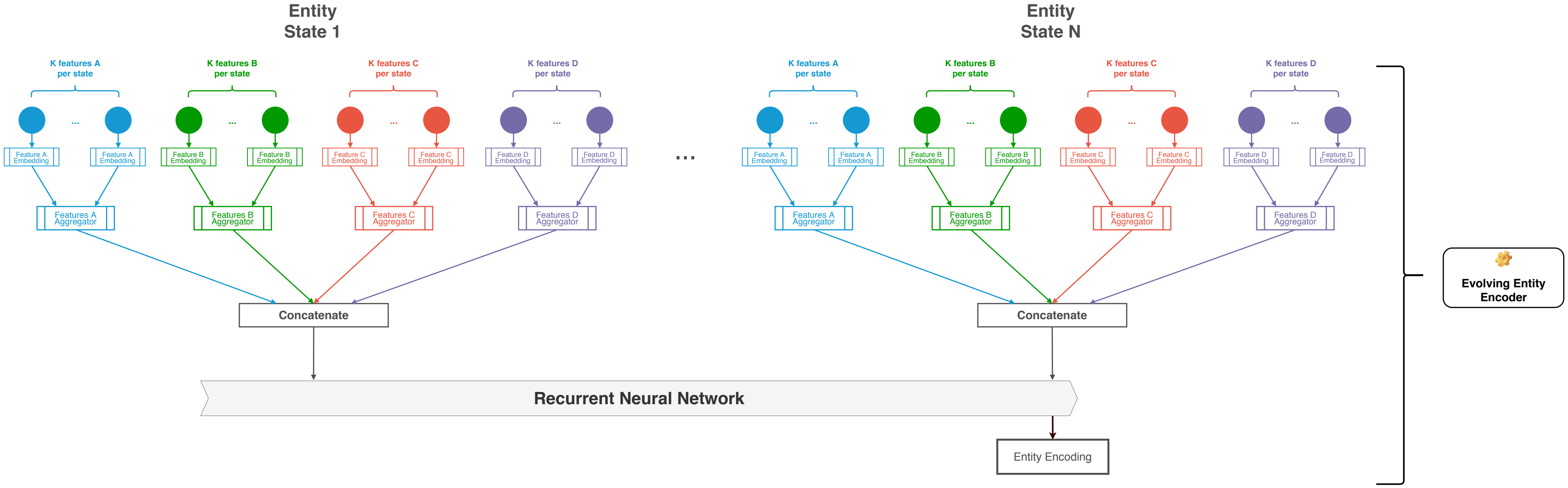


Figure 4.3: Visualization of the *Evolving Entity Encoder*, that passes each feature through their respective embedding, then aggregator and concatenate the results to be finally fed through the Recurrent Neural Network. This is application-agnostic and supports a varying number of features for each entity.

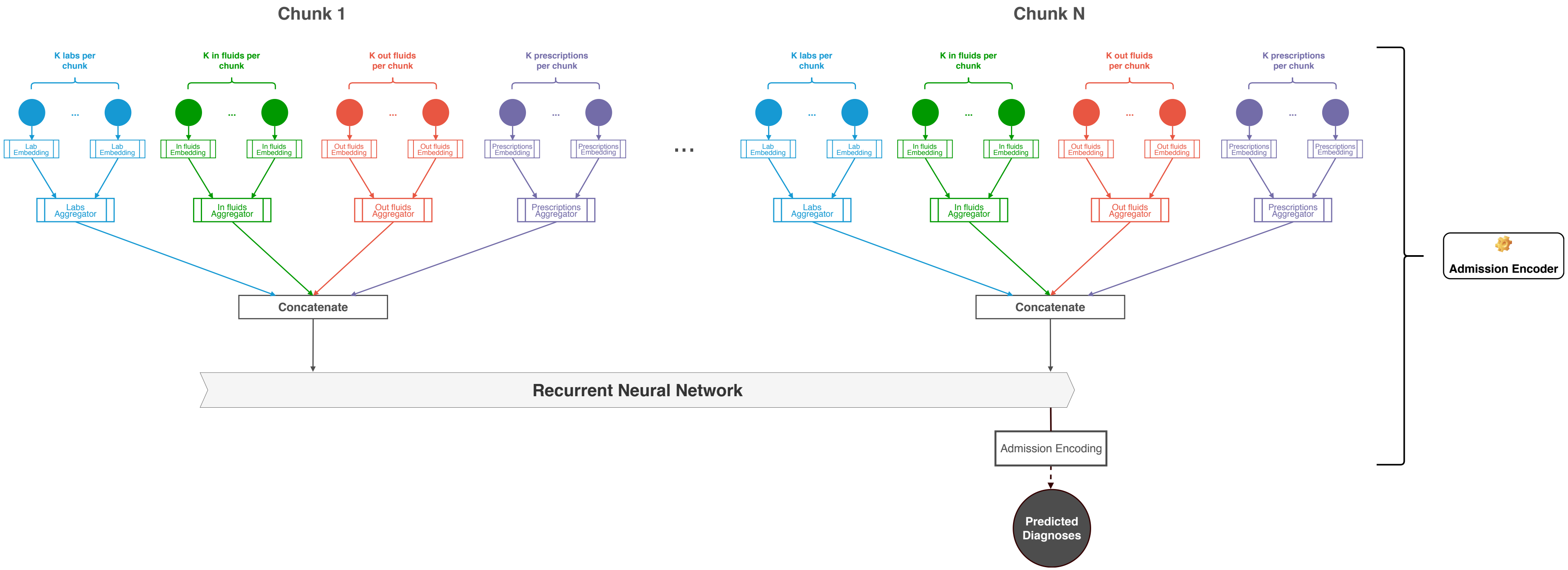


Figure 4.4: The *Evolving Entity Encoder* applied to our use-case. Hence, we are encoding an admission that consists in an evolving patient state to finally predict the diagnoses at discharge with an additional fully-connected layer.

5
KG-RNN: Our Architecture

In this chapter we introduce our own architecture, based on the previous chapter where we present our **Evolving Entity Encoder**. This novel architecture leverage different graph theory techniques and machine learning to encompass information from other entities and thus further boost the predictive power of the model.

The first section will highlight the different steps of the pipeline, then the following section describes the process of constructing the weighted knowledge graph from internal (MIMIC-III) as well as external information. Right after, we develop the idea of "extracting" other entities of interest from the knowledge graph that will improve the prediction for a given input entity. Finally, we expose our *KG-RNN* deep learning architecture to leverage the input entity as well as the ones we extracted in the previous stage.

5.1 Overview

Our "KG-RNN" architecture extends on the previous baseline described in chapter 4. The novelty and improvement rely on neighboring entities of interest (i.e. admissions in our use-case) to encompass more information than solely the input entity.

To do so, the first phase consists in creating an appropriate knowledge graph from our dataset but also external information to enrich the linkage structure between entities. This can be done in many different ways, in a weighted or unweighted fashion ($w = 1$), and we will describe here how we proceeded in the healthcare use-case.

Right after, from this enhanced knowledge graph, weighted or not, we have to extract neighboring entities of interest. For this purpose, we can employ any graph sampling technique (leveraging edge weight or not) and we will go through the one we chose. Finally, from the extracted neighbors and the input entity, we build a machine learning model extending from the baseline one to convey information from these neighbors.

5.2 Weighted Knowledge Graph Construction

As a first step and to the end of building our weighted knowledge graph, the main field of interest is *DIAGNOSIS* (that we will call "prediagnosis" hereafter to avoid confusion with ICD9 diagnoses) in the admission table. Firstly, we clean the prediagnosis: 1. Converting to lower-case. 2. Remove non-alphabetic characters. 3. Remove multi-spaced as well as leading and trailing ones.

On top of this *prediagnosis*, we leverage external information from a recent paper that creates a mapping between diseases and symptoms [18]. This external knowledge graph is built from Electronic Health Records (EHR) and links diseases with their respective symptoms, while providing a symptom relevance weight (between 0 and 1). An example of entry from this external source is: *Migraine: Headache*

($w = 0.384$), nausea ($w = 0.316$), sensitivity to light ($w = 0.223$), ... These diseases and symptoms are cleaned in the exact same way as the prediagnosis and assembled to create our **dictionary**.

Our goal now is to link the external information with our internal information to create an enriched knowledge graph. We want to link prediagnosis with symptoms and diseases, while these latter two are intrinsically linked directly by the external information (cf. *Migrain* example above).

To this end we have to cope with the problem of spelling mistakes as well as quasi-similar prediagnosis and diseases/symptoms (e.g. “Coronary heart disease“ vs. “Arteries heart disease“), we create a character N-grams [3] (where the size of the n-gram has to be tuned) list from the cleaned entries of the dictionary (prediagnosis, disease and symptoms). From this n-grams list, we create a TF-IDF [4] vector with a minimum document frequency of 1.

Finally, we match prediagnosis with diseases and symptoms based on the cosine similarity between their n-grams + TF-IDF vectors. Namely, we link a prediagnosis with a disease and a symptom if their cosine similarity score is above a given threshold (to manually filter out noise), and we link diseases and symptoms based on the external information while also applying a threshold on the symptoms relevance weight provided out-of-the-box by the external source. The final knowledge graph is represented in the figure 5.1, including both internal and external information.

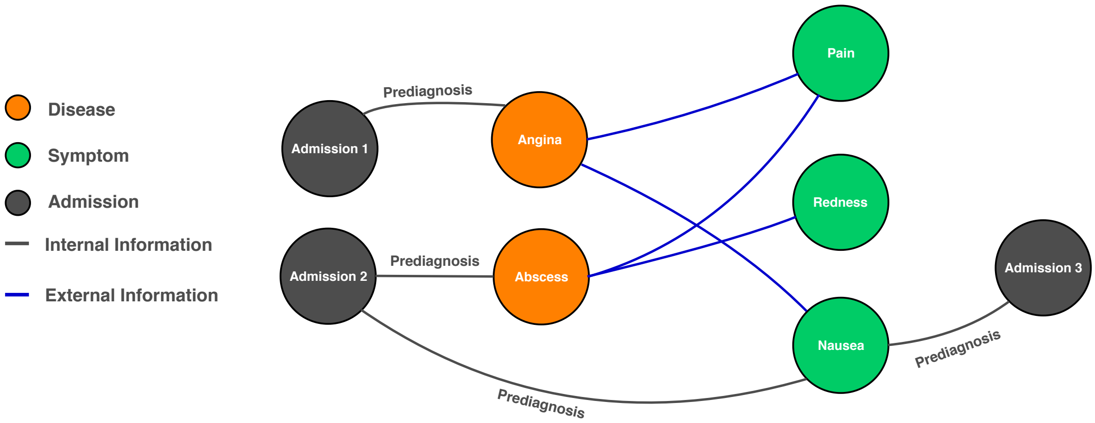


Figure 5.1: Final knowledge graph made of external and internal information that will be used in the experiments and discussions.

On a practical note, for the rest of the discussions and experiments, we set n-grams to **3 characters**, the minimum cosine similarity score of **0.6** and a relevance weight between diseases and symptoms of **0.2**.

[3] https://en.wikipedia.org/wiki/N-gram
[4] https://en.wikipedia.org/wiki/Tf_idf

5.3 Weighted Knowledge Graph Extraction

Secondly, from this weighted knowledge graph, we want to extract relevant neighboring entities to enrich our input entity with additional information. For this purpose, one can use any graph sampling technique [19, 20], making use of the weights defined during *Graph Construction* or not.

This graph sampling will be very important for the downstream model and also the number of neighbors is a critical hyper-parameter that has to be tuned. In that regard, relevant experiments can be found in the appropriate chapter. Some common sampling techniques could be 1. Random sampling on 1-hop neighbors. 2. Snow-Ball sampling. 3. Forest Fire sampling..

For our use-case at hand, we decided to employ some *importance sampling* based algorithm. This technique is inspired from a paper [21] by Pinterest and Stanford, it relies on random walks to create an importance score for each entity and take the top scoring entities as neighbors of interest.

Concretely, the process is to simulate many random walks *starting from input entity* and compute the L_1-normalized visit count as the importance score. Now, instead of simulating thousands of random walks it can be proven that in the limit of an infinity of simulations, the normalized L_1 visit count is equivalent to a Personalized PageRank score. Henceforth, we decided to compute the Weighted Personalized PageRank score (personalized on the input entity) using *Oracle PGX* for each input entity.

Finally, and as stated previously, the top-M scoring entities are extracted for each input admission and we define a minimum score of **0.0001** as an arbitrary threshold. Otherwise, there would always have exactly M neighbors sampled even if some are isolated and thus have a score of 0. This threshold allows for more flexibility and to have $0 \leq$ neighbors $\leq M$ for a given input admission.

From a practical standpoint, we also make sure not to sample neighbors from the testing or validation set when we are training, or respectively sampling entities from the training or testing set during validation. Equivalently, we make sure not to sample entities from training and validation when we do the final evaluation on the test set.

5.4 Graph Machine Learning

The machine learning model encapsulating information from neighboring entities as well as the input entity relies on the building block described in the chapter 4. Indeed, the input admission is first encoded using the **Evolving Entity Encoder** from which the final embedding vector $\tilde{\boldsymbol{h}} \in \mathbb{R}^b$ is fed to the main module. The main module job lies in blending information from extracted neighbors with the input admission vector $\tilde{\boldsymbol{h}}_i^m$.

To better understand the mechanics, the extracted neighbors can be seen as made of static information throughout admission (e.g. patient age) and dynamic information (e.g. their respective events). The main module can either process dynamic information and static information, or just one or the other. Explicitly, the extracted neighbors entities can also go through the Evolving Entity Encoder to embed their dynamic behavior in a vector $\tilde{\boldsymbol{h}} \in \mathbb{R}^b$ that will be then concatenated with static information vectors.

Eventually, the embedded vector is concatenated with static information vectors, and fed through a fully-connected layer (*encoder*) to obtain the final neighbor encoding vector. These vectors are then fed through an aggregator of the same kind as the one described in chapter 4, that is *sum*, *mean* or *max.* hHe purpose of this aggregator is to squash the data along the M dimension to squeeze out information from neighbors. The final prediction diagnoses are made from the concatenation of the aggregated neighbors and input entity encoding, that is finally fed into a fully-connected layer mapping to our *50* classes.

On our use-case at hand, we decided to discard dynamic behavior of neighbors and solely take into account the final diagnoses of those as our static information. Additionally, for a one-hot encoded vector of a neighbor final diagnoses $\boldsymbol{y} = [0, 0, 1, 0, 1, 0, \dots, 1]$, where $|\boldsymbol{y}| = 50$ we decided to make use of the **SEQ_NUM** field by transforming this one-hot vector using the function $1/x$. That is, our new static information vector becomes $\tilde{\boldsymbol{y}} = [0, 0, 1/1, 0, 1/3, 0, \dots, 1/2]$, where $|\tilde{\boldsymbol{y}}| = 50$ if we respectively have a **SEQ_NUM** of 1, 3 and then 2.

A schematic visualization of the general machine learning model is available on the figure 5.2, while the applied model can be found on the figure 5.3.

5.4.1 Schematic Visualizations

In order to provide the reader an overview of the mechanics behind KG-RNN, the following visualization summarizes the textual description of the previous section.

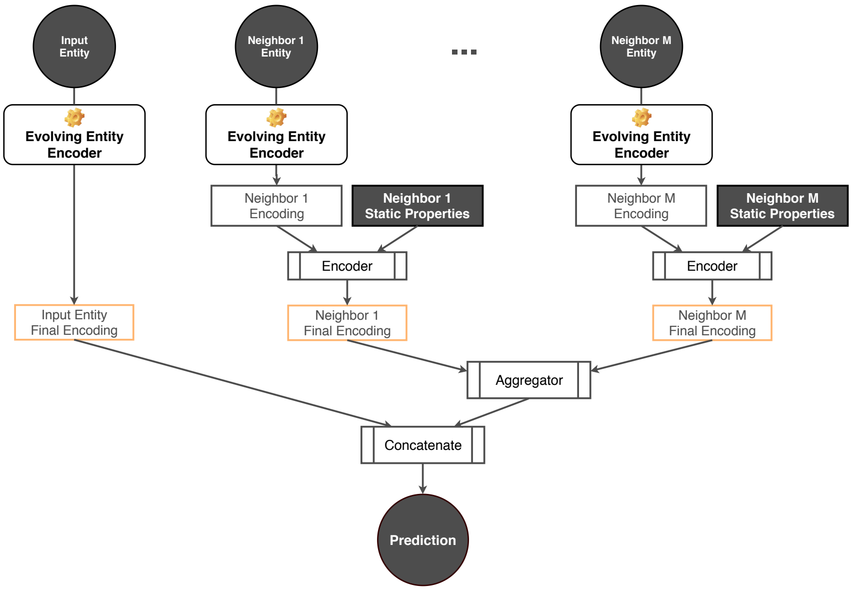


Figure 5.2: The general module of our architecture, where **Evolving Entity Encoder** relies on work from previous chapter. The static properties and information from neighbors are concatenated and fed through an *Encoder*, consisting of a simple fully-connected layers to blend the concatenated information. All encoding vectors from neighbors are aggregated along the M dimension and concatenated with the input entity. This vector is then further fed into a fully-connected layer, here hidden in the *Concatenate* block, to output the predictions.

The following figure is a special case of the general approach available in the previous figure. Indeed, in our practical application we decided to get rid of the dynamic information from neighbors, leading to the following adapted schema:

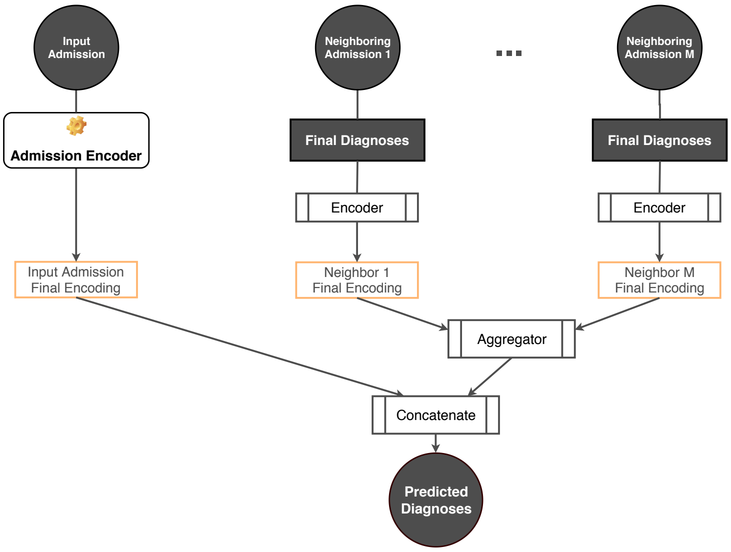


Figure 5.3: Similar to the previous schema, except that only static information from neighbors is encoded and squashed. Here the static information consist of only the diagnoses of these neighboring admissions.

As mentioned previously, in our case we decided to discard dynamic information from neighbors but we argue that this is heavily application-dependent, thus validating on different tasks and knowledge graphs would be insightful.

6 Experiments

This chapter describes the different experiments we ran as well as their respective results. It is heavily focusing on the task at hand, which is to *predict the diagnosed top 50 most frequent ICD9 codes at discharge in a multi-class multi-classification fashion.*

The first section describes the setup in which the experiments are run, the optimizer as well as the hardware setting. Following to this, we get to the core of the chapter and explain our experiments we ran while providing the quantitative results on different metrics. Finally, from these experiments we conclude the chapter with a hyper-parameter tuning section detailing how we picked the deep learning hyper-parameters.

6.1 Setup

All the experiments and training sessions were run on two **Tesla P100** with 16 GB of memory. The CPU was an **Intel Xeon Gold 5120 CPU @ 2.20 GHz**, with **56 cores**, while we had access to **187 GB of RAM** on the machine. On top of that, the different scripts and models have been built on *Python 3.6.7* and we used *Pytorch 1.0.0* as our Deep Learning framework.

We employ mini-batch stochastic gradient descent method together with the Adam optimizer [22] to minimize our multi-class multi-classification loss.

To facilitate reading and understanding, we break down the different hyper-parameters in three groups:

- Knowledge Graph
    - M: number of neighbors
    - Characters n-grams
    - Weighted Personalized PageRank threshold
    - Disease - Symptom threshold
    - Admission - Symptom threshold
    - Admission - Disease threshold
- Admission Processing
    - N: number of chunks
    - K: number of events per chunk
    - τ: chunk length
- Deep Learning
    - n: events embedding dimension (potentially different for each event type)

- a: aggregators hidden dimension (for both input entity and neighbors, potentially different for each event type)
- Input entity aggregator type (max, mean or sum, potentially different for each event type)
- Neighbor entities aggregator type (max, mean or sum)
- RNN: number of layers
- RNN: number of neurons per layer
- RNN: dropout between layers
- RNN: bidirectional or not
- b: Recurrent Neural Network output dimension
- Batch size

To reduce the search space, we contrived our hyper-parameters by fixing the same embedding dimension and aggregator for all events. Finally, we also decided to tune the *Knowledge Graph* and *Admission Processing* hyper-parameters while tuning the *Deep Learning* ones. Consequently, we also did the opposite when tuning the *Knowledge Graph* and *Admission Processing* hyper-parameters.

This way, we limit the computation time while getting a pretty good idea of the top-performing configurations and interactions between theses hyper-parameters. We suppose that scores reported for the final task on the testing set could be further improved by fine-tuning all these hyper-parameters together instead of group-by-group.

Finally, we decided to split the *58'976* in two groups, training and testing set, using an 80%-20% split. We then further holdout 10% of this training set for validation purposes (the training set is thus 72% of the total admissions while validation represents 8% of these).

As a training strategy, we monitor different metrics, as explained in the section 6.2, at each epoch and employ an early-stopping criterion on the validation loss with a patience of 20 epochs. That is, if the validation loss does not decrease within 20 epochs, the model is saved at its best scoring epoch and the training is considered finished.

6.2 Metrics

F1 Score The F_1 score is a particular case of the F_β score which blends the well-known precision and recall scores. The F_β score is computed as follows:

$$F_\beta = (1+\beta)^2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{(\beta^2 \cdot \text{Precision}) + \text{Recall}} \tag{6.1}$$

Basically, the β parameters dictate the importance we want to give to the precision or recall. By setting $\beta = 1$, we give equal importance to both of them and above formula is equivalent to computing harmonic mean between the precision and recall. Our choice of $\beta = 1$ is common and usually a good trade-off when there is no obvious preference between precision and recall.

Area Under Receiver Operating Characteristic Often abbreviated $AUROC$, this metric is coming from the receiver operating characteristic curve that is computed using the “True positive rate“ and “False positive rate“. This curve is particularly interesting since it reports the ability of our classifier to discriminate between positive and negative samples when the threshold varies.

Generally speaking, computing the area under this ROC curve is equivalent to the probability that a classifier will rank a positive random sample **higher** than a random negative one.

Macro and Micro Averages To account for the slight class imbalance depicted in the top-right figure 3.2, we consider two kinds of averages over classes scores, namely *macro* and *micro* averages.

The macro average will compute the metric independently for each class and then average globally, thus treating our different ICD9 codes equally. On the other hand, the micro average will first aggregate the contributions of all classes in order to compute the average metric.

We decided to compute both averages for our two metrics, leading to four combinations that we report in the experiments in sections 6.4:

- Macro-F1 Score
- Micro-F1 Score
- Macro-AUROC
- Micro-AUROC

6.3 Deep Learning Hyper-parameter Tuning

For the hyper-parameter tuning of deep learning parameters, we fixed the *Knowledge Graph* and *Admission Processing* hyper-parameters to the following values:

- Knowledge Graph
  - $M = 10$
  - Characters n-grams = 3 characters
  - Weighted Personalized PageRank threshold = 0.0001
  - Disease - Symptom threshold = 0.2
  - Admission - Symptom threshold = 0.6
  - Admission - Disease threshold = 0.6
- Admission Processing
  - $N = 200$
  - $K = 25$
  - $\tau = 3$ hours

6.3.1 Method

All the scores were computed on the validation set, leaving the testing one untainted to prevent overfitting. The best scoring model was chosen accordingly and in accordance to performances on the validation loss.

We employed a very simple *random search* as a sampling strategy of the next set of hyper-parameters during the optimization. This allows for much better results than a typical grid search that has the tendency to explore only a small subset of the search space. This strategy is also proven to show very good performances when the number of *trials* is limited (≤ 100 in our case) for a very large search space, on top of the ease of implementation compared to more advanced sampling strategies.

6.3.2 Search space

The search space in our case consisted of:

- $n \in \{8, 16, 32, 64, 128\}$, same for all event types
- $a = 64$, same for input entity, neighbors and all event types
- $\text{agg}_{\text{input}} \in \{\text{max}, \text{mean}, \text{sum}\}$, same for all event types
- $\text{agg}_{\text{neighbors}} \in \{\text{max}, \text{mean}, \text{sum}\}$, same for all event types
- $\text{RNN}_{\text{layers}} \in \{1, 2, 3, 4, 5\}$
- $\text{RNN}_{\text{neurons}} \in \{64, 128, 256, 512\}$
- $\text{RNN}_{\text{dropout}} \in \{0, 0.1, 0.25, 0.5\}$
- $\text{RNN}_{\text{bidirectional}} \in \{\text{True}, \text{False}\}$
- $\text{RNN}_{\text{output size}} = 64$
- Batch size $\in \{4, 8, 16, 32, 64, 128\}$

Which leaves us with a search space of $5 \times 3 \times 3 \times 5 \times 4 \times 4 \times 2 \times 6 = 43'200$ possibilities.

6.3.3 Results

Baseline We obtained the following best-scoring combination of hyper-parameters for the baseline:

- $n = 64$
- $a = 64$
- $\text{agg}_{\text{input}} = \text{mean}$
- $\text{RNN}_{\text{layers}} = 1$
- $\text{RNN}_{\text{neurons}} = 128$
- $\text{RNN}_{\text{dropout}} = 0.25$
- $\text{RNN}_{\text{bidirectional}} = \text{True}$
- $\text{RNN}_{\text{output size}} = 64$
- Batch size = 16

Which translates to a particularly lightweight model, indicating the tendency of the baseline to overfit rather quickly.

KG-RNN We obtained the following best-scoring combination of hyper-parameters for the baseline:

- $n = 16$
- $a = 64$
- $\text{agg}_{\text{input}} = \text{max}$
- $\text{agg}_{\text{neighbors}} = \text{mean}$
- $\text{RNN}_{\text{layers}} = 2$
- $\text{RNN}_{\text{neurons}} = 128$
- $\text{RNN}_{\text{dropout}} = 0.5$
- $\text{RNN}_{\text{bidirectional}} = \text{False}$
- $\text{RNN}_{\text{output size}} = 64$
- Batch size = 128

It is important to notice that this top-scoring configuration consists in using a different kind of aggregator for neighboring admissions and input events.

One could argue that using a *max-pooling* for input events and *mean-pooling* for neighbors final diagnoses make sense since we can hypothesis that most salient events are important for a single patient, while blending the final diagnoses of neighbors allows to account for their relative contributions.

6.4 Experiments

Each experiment is broken down in a short description of the experiment and the associated results and graphics, followed by a “discussion“ section where we interpret the results and plots.

All of these are on validation set for sake of keeping the testing set untainted until the final evaluation of the task at hand, namely ICD9 code prediction. Therefore, results of the different metrics, presented in sections 6.2, are computed on the validation set for all experiments and compared to the baseline scores on each figure.

For all these experiments, the base setup of hyper-parameters are the one from the previous section for the *Deep Learning* group. For the two other groups, here are the default parameters we chose:

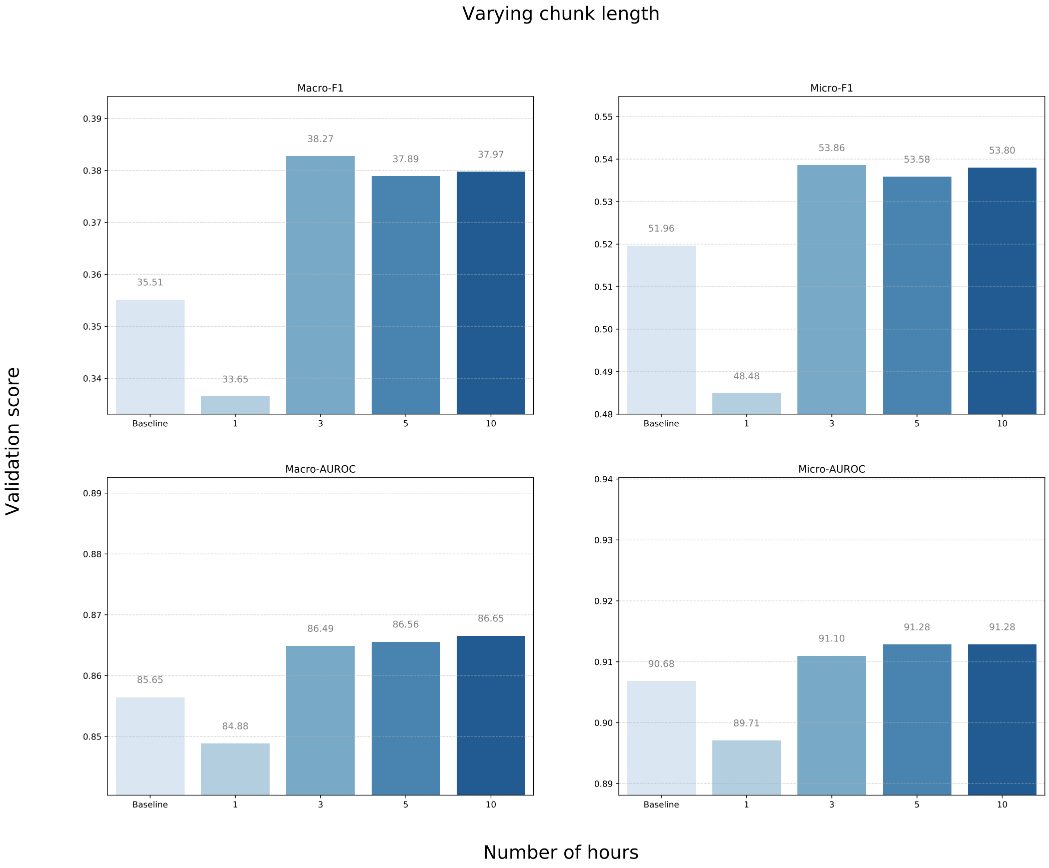
- Knowledge Graph
  - $M = 10$
  - Characters n-grams = 3 characters
  - Weighted Personalized PageRank threshold = 0.0001
  - Disease - Symptom threshold = 0.2
  - Admission - Symptom threshold = 0.6
  - Admission - Disease threshold = 0.6
- Admission Processing
  - $N = 200$
  - $K = 25$
  - $\tau = 3$ hours

These are the hyper-parameters to be assumed except for the varying one in each experiment, that will be reminded in the relevant subsection. It is important to bear in mind that complex interactions within hyper-parameters are not taken into account in these experiments since we tune one degree of freedom at a time.

It would be very interesting to run these experiments with more computational power and time to account for much more precise hyper-parameter tuning and visualize how the metrics react for possibly more complex combinations.

6.4.1 Chunk length

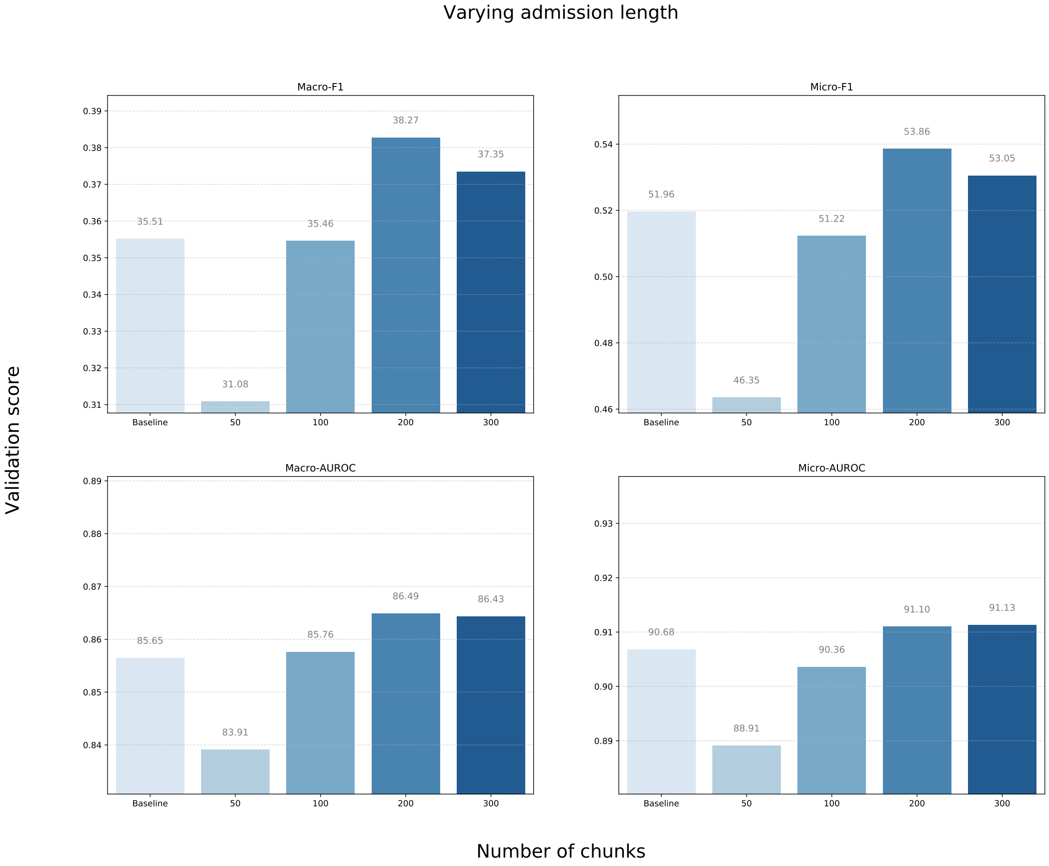
In this experiment we make the number of hours per chunk vary. This means that we consider the larger time window as the current graph state. Thus, it should have an impact on the amount of information fed to *KG-RNN*, especially its aggregators.



Discussion We see that the number of hours per chunk does not have a huge impact after 3 hours. However, setting the chunk length to small (i.e. 1 hour) makes the graph state probably too sparse in terms of the number of events.

6.4.2 Admission Length

This experiment makes the number of chunks per admission vary, also referred as the number of graph states. This will have the impact of increasing the size of sequences that are fed to the downstream Recurrent Neural Network and challenge its capacity to encode long-term dependencies in the input sequences.

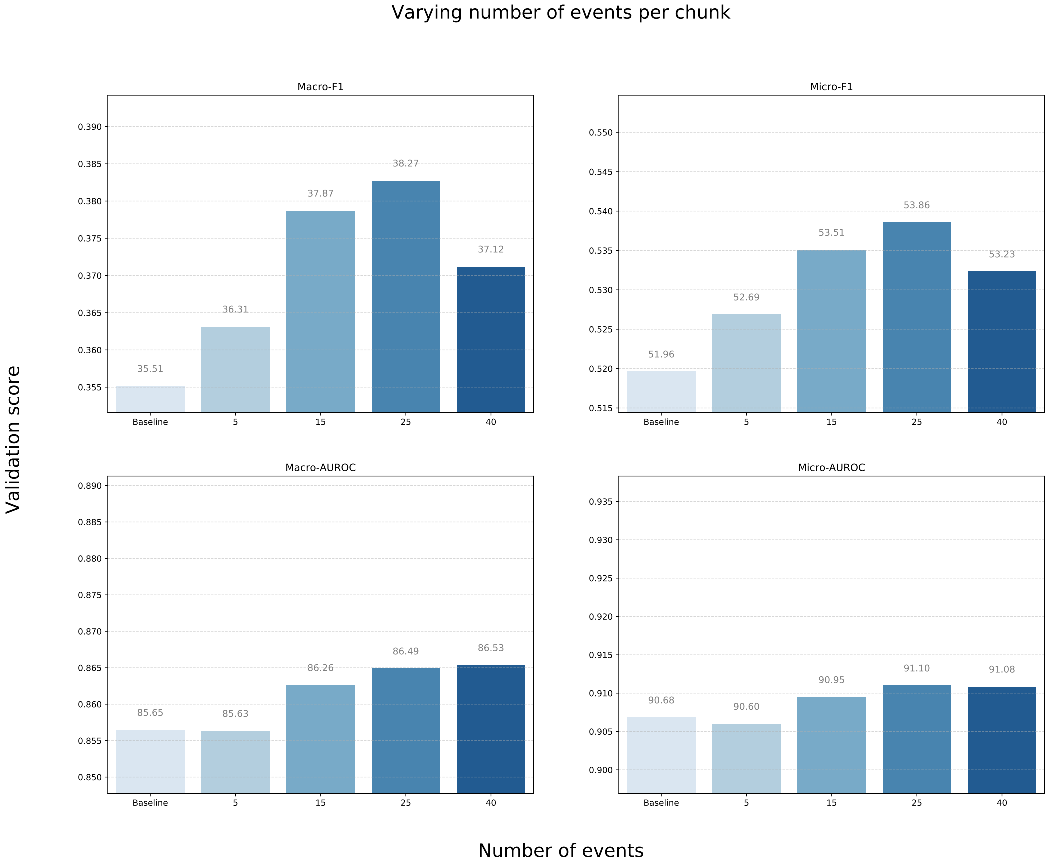


Discussion In the above figure, we clearly see that we obtain a local optimum with an admission length of *200* for the F1 score. However, the optimal value for the AUROC score is not so clear cut and further investigation would be beneficial in terms of interaction with the **chunk length**.

Indeed, since the total considered time for an admission is chunk length × admission length, both parameters are intrinsically linked and probably heavily correlated.

6.4.3 Number of events per chunk

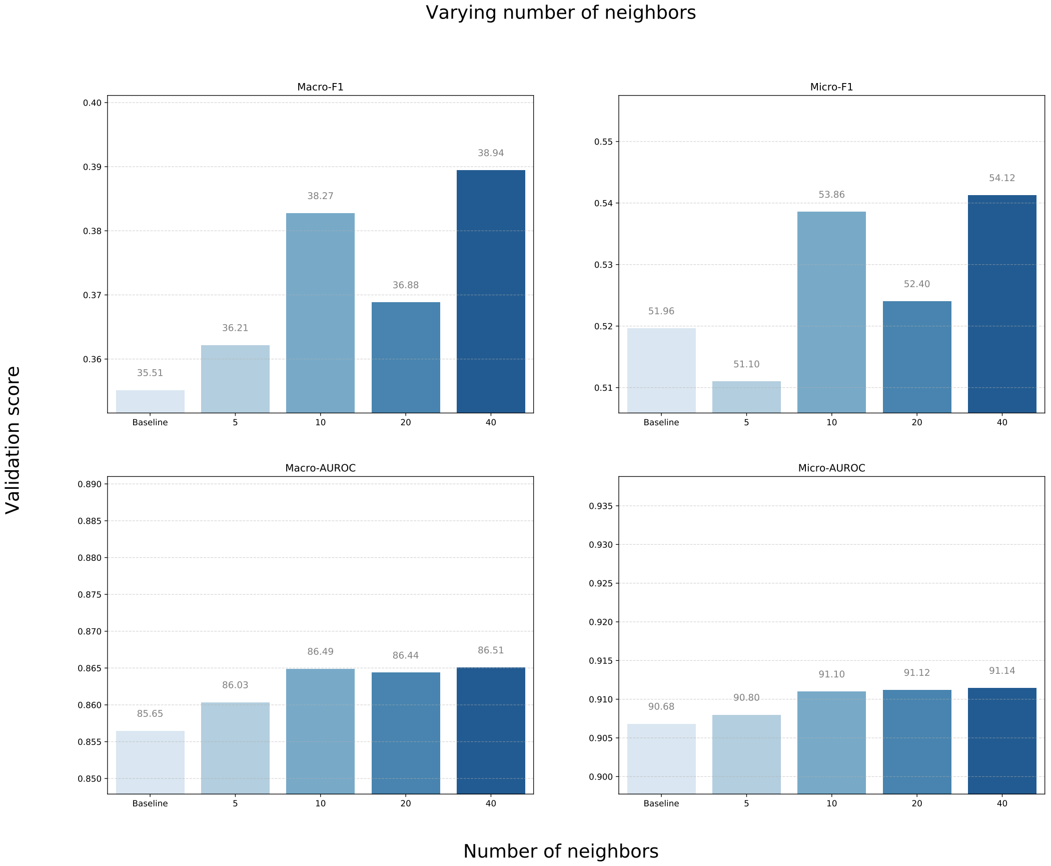
For this experiment, we vary the *maximum* number of events per chunk and per type. Chunks that have more than this number of events in a chunk for a given type will be randomly sampled to K, while other ones remain with their number of events $\leq K$.



Discussion As in the previous experiment, we see a clear optimum for the F1 score at 25 events per chunk and per type. Again, the conclusion is not so obvious for the AUROC score, and this metric does not seem to improve much above $K = 15$.

6.4.4 Number of Neighbors

As a last experiment, we made the number of neighbors for each input entity varies. As the previous experiment, this defines the maximum number of neighbors and does not mean it will have exactly M neighbors. Indeed, it is important to remember that we set a minimum $WPPR$ threshold of 0.0001 that might lead to some isolated admissions having $\leq M$ neighbors.



Discussion The results of this experiment are very interesting, yet difficult to conclude on. We can see that the number of neighbors has a huge impact on the F1 metrics, but not so much on the AUROC score. However, it is important to notice that we would expect the bar plots to have a similar shape as before.

That is, we would expect it to have a concave shape, where fewer neighbors means less information and thus decreased scores. Whereas, too many neighbors would add more noise than information and also hindering results. Therefore, we would expect to find an optimum value in-between but we see a drastic decrease between *10* and *20* neighbors, while having an increase in our metrics between *20* and *40* neighbors. Lastly, the scores with *40* neighbors seem slightly better than with *20*.

6.5 ICD9 Prediction

6.5.1 Quantitative analysis

From these top-scoring hyper-parameters for the different groups, we selected the following parameters and ran our final evaluation for the task of predicting the top 50 ICD9 codes in a multi-class multi-classification scenario.

These results are all performed on the **testing set**, after all the previous hyper-parameter tuning and experiments were run on the validation set. To this end, we left the testing set untainted and the results of this section and for the task at hand will be as representative as possible from a real scenario. This testing set is made of 20% of the dataset, totaling *11'230* admissions.

The two following plots are interesting to compare between the baseline performances and our *KG-RNN*:

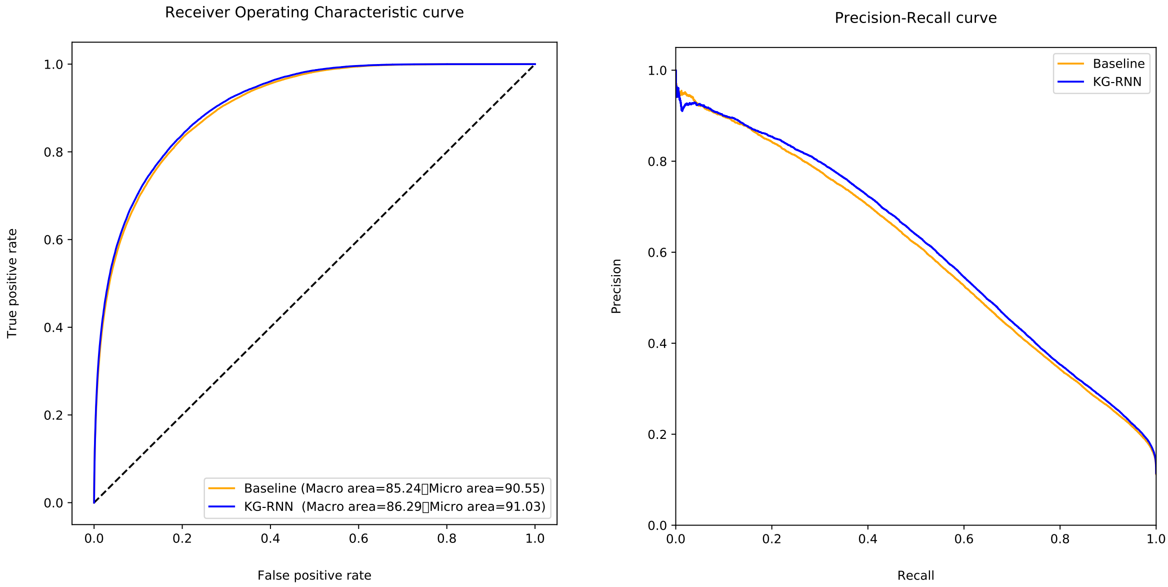


Figure 6.1: **Left**: Receiver operating characteristic curve on the testing set for both the *baseline* and *KG-RNN*. **Right**: Precision-recall curve on the testing set for both the *baseline* and *KG-RNN*.

We also evaluated the results on the different metrics that we described in the section 6.2:

Metric	**Average**	**Model**	**Score**
F1	Macro	Baseline KG-RNN	**36.52%** **37.90%** (+1.38%)
	Micro	Baseline KG-RNN	**51.55%** **53.47%** (+1.92%)
AUROC	Macro	Baseline KG-RNN	**85.24%** **86.29%** (+1.05%)
	Micro	Baseline KG-RNN	**90.55%** **91.03%** (+0.48%)
Accuracy	-	Baseline KG-RNN	**92.22%** **92.36%** (+0.14%)

Discussion We notice an improvement on each metric for both averages. Yet, the results are not easily distinguishable and very close to each other on the testing set compared to the validation set where the improvements from *KG-RNN* were stronger.

To further assess the quality of our *KG-RNN* model, we decided to use a statistical significance test, the McNemar's Test [5]. Indeed, this test will allow us to get a very precise idea of the relevance of the edge that *KG-RNN* seems to have over the baseline.

The McNemar's test requires to build a "contingency table" as follows from our results:

	KG-RNN is correct	**KG-RNN is incorrect**
Baseline is correct	510'590 (a)	7'247 (b)
Baseline is incorrect	7'984 (c)	35'679 (d)

If we consider p_b and p_c as the theoretical cell probabilities, the null hypothesis and the alternative are:

$$H_0 : \quad p_b = p_c$$
$$H_1 : \quad p_b \neq p_c$$

In other words, **under the null hypothesis, the two models should have the same error rate.** and we can summarize this as follows:

- **Fail to Reject Null Hypothesis**: Classifiers have a similar proportion of errors on the testing set.
- **Reject Null Hypothesis**: Classifiers have a different proportion of errors on the testing set.

Where under the null hypothesis and with a sufficient amount of data (which is our case), the statistic should follow a chi-squared distribution with 1 degree of freedom.

Notably, in our case the *Corrected McNemar* test statistic is:

$$\chi^2 = \frac{(|b-c|-1)^2}{b+c} = \frac{(|7247-7984|-1)^2}{7247+7984} = 35.57 \tag{6.2}$$

and the associated **p-value** under H_0 is $P = 0.0000000025$, which is extremely small and statistically significant.

Hence, our p-value shows sufficient evidence to reject the null in favor of the alternative hypothesis. Therefore, the two models have different performances when trained on this particular testing set, supporting the idea that the edge over the baseline is not due to randomness but rather a truly improved predictive power.

[5] https://en.wikipedia.org/wiki/McNemar's_test

6.5.2 Qualitative Analysis

Embeddings First off, we decided to analyze the embedding we optimized while training *KG-RNN*. For this purpose, we projected our embedding matrices into two dimensions using a Uniform Manifold Approximation and Projection [23]. Then, we picked a laboratory measurement (**White Blood Cells**) and a prescription (**Sodium**) to extract their nearest neighbors in the projected 2D manifold:

Neighbors for **White Blood Cells**

Rank	**Laboratory measure**
1	WBC
3	White Cells
6	Immunoglobulin A

Neighbors for **Sodium**

Rank	**Prescription**
1	Sodium Chloride Nasal
2	Sodium Chloride 0.9% Flush
5	Famotidine

In the first case (i.e. *White Blood Cells*), it is very interesting to notice that among the neighbors, we first find an abbreviation of the same laboratory measurement: **WBC**, as well as a short name of it at the third-closest neighbor: **White Cells**. Finally, we also see the compelling evidence that the embedding is of good quality by the positioning of **Immunoglobulin A**, which are antibodies produced by white blood cells.

In the second table, we see that *Sodium* is close to its related prescriptions in the first two nearest neighbors. Whereas, **Famotidine** is also ranked very close at the fifth rank, and it is known in the medical field that Famotidine is often given in **0.9% Sodium Chloride** through intravenous, our second neighbor.

Overall, we can see with these two examples that *KG-RNN* has not only learned similar events, such as "White Blood Cells" and "WBC", but also intrinsically and correlated linked events. Indeed, as we noticed, our deep learning architecture mapped events close to each other in the embedding space if they are related in the medical domain, learning more complex interactions.

Predictions Secondly, to have a better understanding of the value provided by *KG-RNN* over the baseline, we extracted a few samples where the baseline was incorrect but KG-RNN was right in its prediction (represented by c in the "contingency table").

From the extracted samples and to the purpose of understanding the differences, we also checked the information conveyed by neighbors of the input admission. To this end, we are able to analyze and visualize where the correctness of *KG-RNN* over the baseline comes from.

In the following graphs, the first row corresponds to the input admission while the other ones to its neighbors. Each red line shows an ICD9 code actually diagnosed at discharge (i.e. the ground truth), and the horizontal gray line represents the default 0.5 threshold above which a prediction is considered. In the input admission graph (first row), the orange and blue bars are the estimated probabilities by the models, respectively the baseline and *KG-RNN*. Finally, in the neighbor rows, the dark-gray bars represent their final diagnoses (i.e. entity static information) as one-hot vectors.

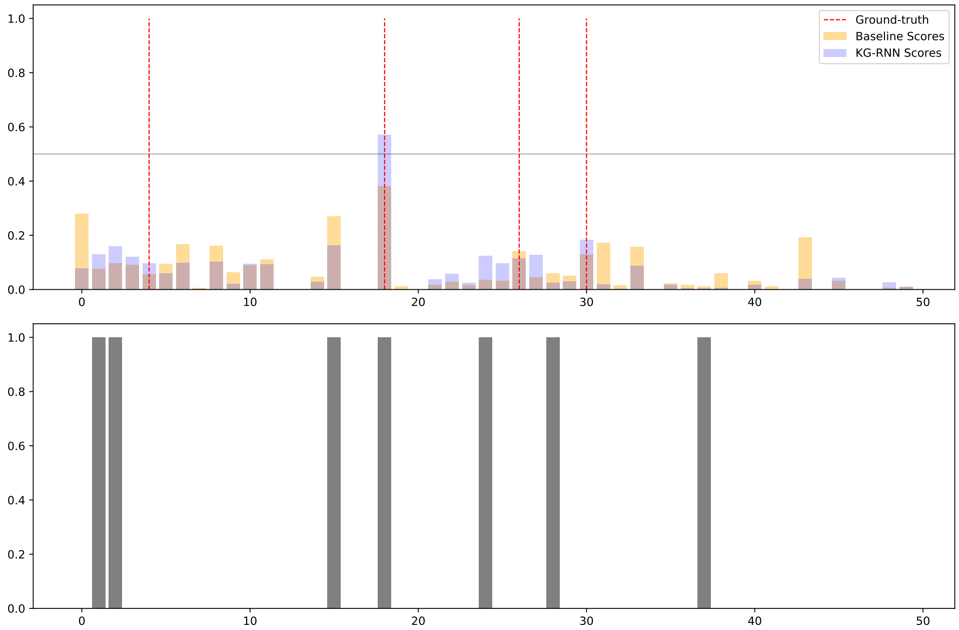


Figure 6.2: In this plot we can see that the neighbors provide the necessary information to *KG-RNN* to improve over the baseline. Indeed, this model does not detect any diagnostic, whereas KG-RNN is able to predict the second one with the help of its neighbor. Namely, we hypothesize that since its neighbor static information conveys the ICD9 code of interest (the second one), it helped pushing the confidence of KG-RNN upward, above the 0.5 decision threshold.

The ICD9 codes of the ground truth and the neighbor from the previous figure are summarized in the following table:

Input admission

Rank	ICD9 Code	Description
4	272.4	Other and unspecified hyperlipidemia
18	401.9	Unspecified essential hypertension
26	486	Pneumonia, organism unspecified
30	530.81	Esophageal reflux

Neighbor

Rank	ICD9 Code	Description
1	244.9	Unspecified acquired hypothyroidism
2	250.00	Diabetes mellitus without mention of complication, type II or unspecified type, not stated as uncontrolled
15	38.93	Unspecified septicemia
18	401.9	Unspecified essential hypertension
24	427.31	Atrial fibrillation
28	507.0	Pneumonitis due to inhalation of food or vomitus
37	96.6	Late syphilis, latent

It is interesting to see that even if the neighbor diagnoses do not match the ground truth *perfectly*, some ICD9 codes still convey information for the input admission. Indeed, if we look at the number **26** in the input admission (i.e. *Pneumonia*), it could be linked to the number **28** of the neighbor (i.e. *Pneumonitis*). However, even if the final diagnoses are different ("Pneumonitis" indicates inflammation of lung tissues, whereas "Pneumonia" is inflammation caused by an infection), the admissions are linked through their pre-diagnosis and thus one may carry information on the other.

Therefore, using the ICD9 code hierarchy may help the model to make sense from neighbors diagnoses and increase predictive power for the input admission. This potential trail is further described in the appropriate section 7.2 of the last chapter.

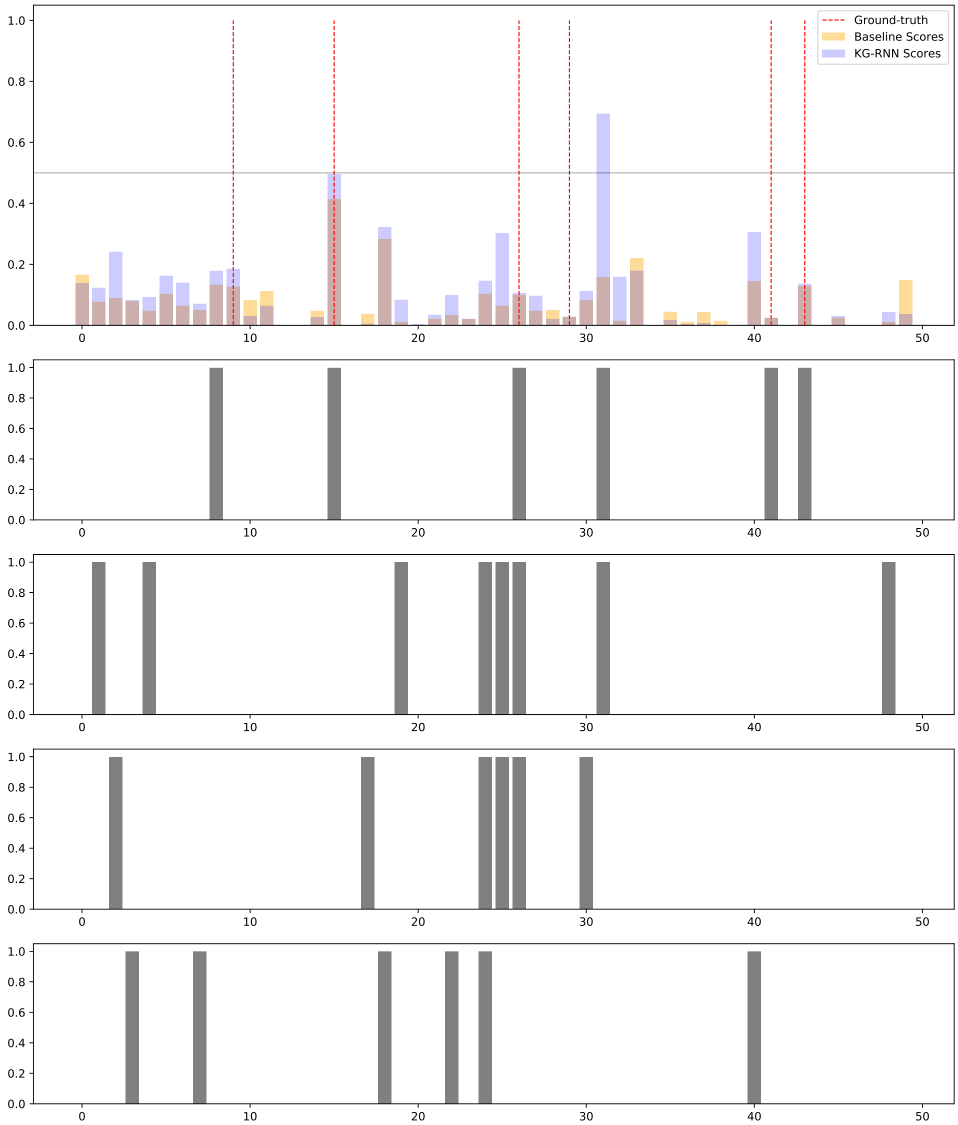


Figure 6.3: In this plot, the input admission has 4 neighbors with different final diagnoses (static information), and we also notice interesting behavior of *KG-RNN* compared to the baseline. In this scenario, we see that the first neighbor helps pushing the confidence of the model over the decision threshold for the second diagnosed ICD9 code. However, we also see that *KG-RNN* is misled by the first two neighbors that are wrongly pushing ICD9 number thirty-one over the decision threshold. This highlights the importance of a good knowledge graph construction and neighbors extraction algorithm.

To conclude, we see that neighbors are having a big impact on KG-RNN, as the predictions where $\text{Score}_{\text{baseline}} \geq \text{Score}_{\text{KG-RNN}}$ occur when none of the neighbors have a matching ICD9 code. On the other hand, when one or more of the neighbors have a certain ICD9 code, the corresponding score for *KG-RNN* on the input admission is consistently pushed upwards. This reveals how much KG-RNN learns to rely on its neighbors to score its predictions.

7 Concluding Remarks

In this concluding chapter, we first go through a conclusion of the work presented in this thesis, in the section 7.1. We summarize our results and discuss the potential answer to the questions in the section 1.3.

Finally, we elaborate on some potential trails for future work and how the current version of *KG-RNN* could be improved. Each idea of possible improvement is described briefly in a separate paragraph and were not implemented mostly due to time constraints.

7.1 Conclusion

In this thesis, we proposed a novel architecture and pipeline for processing evolving knowledge graphs. This architecture has shown to exploit well information from other entities, through complex linkage structure created with external data sources.

For our specific use-case in health-science, our enriched knowledge graph has been processed to naturally cope with many of the challenges inherent to the application. In particular, we proposed a way to create *patient states* from the knowledge graph that can be seen as *graph states* and further fed into our KG-RNN pipeline.

In this scenario of predicting the top-50 diagnoses at discharge, our solution shows significant improvement over a simple single-admission setting. We evaluated our results both quantitatively and qualitatively to understand the pros and cons of the approach. These bottlenecks helped suggest further improvements that could be experimented to design a better second version of the *KG-RNN* pipeline.

Lastly, it would be very insightful to test our architecture on other use-cases in order to cross-validate qualitative and quantitative performances. More over, understanding the role of the different hyper-parameters and their impact on other use-cases could lead to beneficial insights for future versions of the pipeline.

7.2 Future Work

As explained previously, this section is subdivided in paragraphs that each explains a potential improvement over our proposed pipeline.

These suggestions are not ordered in terms of priority and may be implemented in whatever order the reader prefers, as they are all independent from each other.

Knowledge Graph Extensions The first and most obvious improvement lies in adding external sources to our knowledge graph to enrich the information it may carry. We hypothesize that our *Weighted Personalized PageRank* approach scales well and makes even more sense for complex linkage structures.

This improvement may lead to important changes in the different metrics as our KG-RNN pipeline seems to leverage the knowledge graph information.

Events sampler Within the data formation section 3.2.2, we describe that if the number of events in a given chunk for a certain type exceeds K, we randomly sample those.

A potential improvement would be to use a more advanced sampling techniques to down-sample events with respect to their importance. Indeed, some events might carry more information for the final diagnoses than others and they could be discarded by *blindly* sampling randomly among them.

Neighbors Weighting In the current pipeline proposal, all neighbors are considered of equal importance. However, this assumption is incorrect as some may be more relevant than others with respect to the input admission.

To account for neighbor importance, one could use the WPPR score as a weighting coefficient for each neighbor, in front of the encoding vector, for example. However, this improvement will be more relevant as the knowledge graph structure complexity increases, leading to disparate coefficients, as with the current knowledge graph they are pretty uniforms among sampled neighbors.

Smarter Hyper-Optimization A better hyper-parameter optimization is a very interesting low-hanging fruit to explore. Indeed, given the number of hyper-parameters and limited computational power as well as time constraints, a simple random search is likely to miss top-scoring configurations.

On the other hand, something like “Bayesian optimization“ or “TPE optimization“ could be smarter in how they explore the hyper-parameter space and find a better setting much faster. This improvement would be very easily implemented and is independent from models or pipeline improvements.

Distance-based Aggregators If we increase the number of sources of information for the knowledge graph, its complexity will inherently increase. That is, the WPPR-based sampling will most likely sample neighbors at different hop distance.

In that regard, it may be interesting to adopt a strategy like GraphSAGE [17] which builds a different aggregator for each hop distance. This strategy could be easily coupled with our approach to have a different aggregator for each entity type (i.e. event type).

Threshold Tuning For any practical application, the final threshold to pass before triggering an alert is a critical parameter. Indeed, this threshold is very application dependent and rules the precision vs.

recall trade-off.

This threshold tuning could be automated or semi-automated by using a testing set to define the optimal threshold given precision or recall level we would like to reach.

Chunks Overlap During the data preparation described in section 3.2, we could have an additional parameter to calibrate how much overlap there should be between two consecutive chunks. Currently, the KG-RNN pipeline does not add any overlap between chunks and they are perfectly disjoint and defined by τ.

However, overlapping patient states may convey more and smoother information than disjoint ones, at the cost of adding an additional hyper-parameter (that could be trivially set to $\tau/2$). We argue that this need of overlapping chunks vs. disjoint ones is application dependent and hence a hyper-parameter may suit better to test different configurations for the task at hand.

ICD9 Hierarchy As a last second application dependent improvement (on top of “Admissions Chunking“), and as discussed in the qualitative study of our results in subsection 6.5.2, it would be very valuable to account for medical domain knowledge. In that purpose, leveraging information from the ICD9 hierarchy could lead to improved results and interpretability of neighbors.

Indeed, in the current version of the knowledge graph, two neighbors are indirectly linked together through their pre-diagnosis. This information may be similar between two admissions but lead to different diagnoses. However the current architecture does not account for **diagnoses distance**. As an example, *Pneumonia* and *Pneumonitis* are probably “closer“ in relevance than *Pneumonia* and *Diabetes*, and the ICD9 hierarchy may help to convey this kind of information with some tree-based distance. Practically, distances between diagnoses could be incorporated in the knowledge graphs by creating additional links between admissions, weighted by their distance in the ICD9 hierarchy.

In a similar fashion, the loss could be adapted to decrease impact of errors where the distance within the tree is smaller. As an example, the two codes “401.9“ and “401.8“ are harder to distinguish than “401.9“ and “507.8“, and thus the model should be less *punished* for mistaking the first two.

Bibliography

[1] G. S. Halford, R. Baker, J. E. McCredden, and J. D. Bain, “How many variables can humans process?,” *Psychological Science*, vol. 16, no. 1, pp. 70–76, 2005. PMID: 15660854.

[2] M. Minsky, “A framework for representing knowledge,” tech. rep., Cambridge, MA, USA, 1974.

[3] R. Davis, H. E. Shrobe, and P. Szolovits, “What is a knowledge representation?,” *AI Magazine*, vol. 14, pp. 17–33, 1993.

[4] L. Y. X. R. Liu Zhiyuan, Sun Maosong, “Knowledge representation learning: A review,” *Journal of Computer Research and Development*, vol. 53, no. 2, p. 247, 2016.

[5] M. Nickel, K. Murphy, V. Tresp, and E. Gabrilovich, “A review of relational machine learning for knowledge graphs,” *Proceedings of the IEEE*, vol. 104, pp. 11–33, Jan 2016.

[6] M. JORDAN, “Attractor dynamics and parallelism in a connectionist sequential machine,” *Proceedings of the Ninth Annual conference of the Cognitive Science Society*, pp. 531–546, 1986.

[7] B. A. Pearlmutter, “Learning state space trajectories in recurrent neural networks,” *Neural Computation*, vol. 1, pp. 263–269, June 1989.

[8] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[9] K. Cho, B. van Merrienboer, Ç. Gülçehre, F. Bougares, H. Schwenk, and Y. Bengio, “Learning phrase representations using RNN encoder-decoder for statistical machine translation,” *CoRR*, vol. abs/1406.1078, 2014.

[10] J. Chung, Ç. Gülçehre, K. Cho, and Y. Bengio, “Empirical evaluation of gated recurrent neural networks on sequence modeling,” *CoRR*, vol. abs/1412.3555, 2014.

[11] T. Jiang, T. Liu, T. Ge, L. Sha, S. Li, B. Chang, and Z. Sui, “Encoding temporal information for time-aware link prediction,” in *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pp. 2350–2354, Association for Computational Linguistics, 2016.

[12] C. Esteban, V. Tresp, Y. Yang, S. Baier, and D. Krompaß, “Predicting the co-evolution of event and knowledge graphs,” in *2016 19th International Conference on Information Fusion (FUSION)*, pp. 98–105, July 2016.

[13] R. Trivedi, H. Dai, Y. Wang, and L. Song, “Know-evolve: Deep reasoning in temporal knowledge graphs,” *CoRR*, vol. abs/1705.05742, 2017.

[14] Q. Wang, Z. Mao, B. Wang, and L. Guo, “Knowledge graph embedding: A survey of approaches and applications,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 29, pp. 2724–2743, Dec 2017.

[15] S. Kumar, X. Zhang, and J. Leskovec, “Learning dynamic embeddings from temporal interactions,” *CoRR*, vol. abs/1812.02289, 2018.

[16] Z. Che, S. Purushotham, K. Cho, D. Sontag, and Y. Liu, “Recurrent neural networks for multivariate time series with missing values,” *Scientific Reports*, vol. 8, no. 1, p. 6085, 2018.

[17] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," *CoRR*, vol. abs/1706.02216, 2017.

[18] M. Rotmensch, Y. Halpern, A. Tlimat, S. Horng, and D. Sontag, "Learning a health knowledge graph from electronic medical records," *Scientific Reports*, vol. 7, no. 1, p. 5994, 2017.

[19] P. Hu and W. C. Lau, "A survey and taxonomy of graph sampling," *CoRR*, vol. abs/1308.5865, 2013.

[20] J. Leskovec and C. Faloutsos, "Sampling from large graphs," in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, (New York, NY, USA), pp. 631–636, ACM, 2006.

[21] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph convolutional neural networks for web-scale recommender systems," *CoRR*, vol. abs/1806.01973, 2018.

[22] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014.

[23] L. McInnes, J. Healy, and J. Melville, "UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction," *ArXiv e-prints*, Feb. 2018.