POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

A study of possible improvements in Knowledge Tracing with Natural Language Processing and self-attention

Tesi di Laurea Magistrale in
Computer Science and Engineering - Ingegneria Informatica

Author: **Simone Sartoni**

Student ID: 10583763
Advisor: Prof. Paolo Cremonesi
Co-advisors: Luca Benedetto, PhD
Academic Year: 2021-2022

Abstract

Knowledge tracing is the task of modelling the evolution of students' knowledge over time while they answer a sequence of exercises. In online education systems, predicting the answers correctness of a student can provide a more valuable and personalized learning process. Recent works have proposed self-attention based models or considered the text of the exercises to learn additional relations between them. Those areas are still under-explored, so we focus on them. First, we propose Prediction Oriented Self-attentive knowledge Tracing (POST), a self-attention based model composed of: an encoder, embedding past exercises content information, a decoder, adding past answers and a second decoder comparing the outputs with the content information of the target exercise. Then we examine six Natural Language Processing methods (i.e. CountVectorizer, word2vec, doc2vec, DistilBERT, Sentence Transformer and BERTopic) to produce embeddings from the exercises' texts and develop "NLP-enhanced" versions of Deep Knowledge Tracing and POST, able to use these embeddings as inputs. Furthermore, we study how to create hybrid models, able to use at the same time the embeddings produced by multiple NLP methods. We present two hybrid approaches for NLP-DKT. We evaluate the models on three public datasets (*ASSISTments 2009*, *ASSISTments 2012*, *Peking Online Judge*) and on a private dataset provided by *Cloud Academy*. The results show that POST achieves state-of-the-art performance in knowledge tracing with large improvements in binary accuracy and AUC (up to 6.65% and 7.74% respectively) compared to both SAINT+ and DKT. NLP-enhanced DKT has shown to improve DKT results on the three public datasets, while NLP-enhanced POST outperforms the AUC of POST on each of the four datasets. In particular, NLP-POST provides significant improvements on the datasets with few samples. Lastly, hybrid models, implemented only for NLP-DKT, enhance further the results, establishing itself as the model with the highest AUC on *ASSISTments 2009* dataset, improving by 8.73% the AUC of DKT. We conclude that NLP-enhancing and new self-attention architectures are successful directions to improve KT. For future research, we suggest studying and optimizing how to NLP-enhance self-attention models.

Keywords: Knowledge Tracing; Natural Language Processing; deep learning; self-attention; Transformer, answer correctness prediction

Sommario

Nel contesto dell'apprendimento online, è utile tracciare l'evoluzione delle conoscenze e abilità degli studenti nel corso del tempo, col fine di ottimizzare l'utilità degli esercizi proposti. In particolare, prevedere la correttezza delle risposte ad esercizi futuri permette di personalizzare e aggiungere valore al processo di apprendimento, per esempio suggerendo esercizi mirati a colmare eventuali lacune. Recentemente vi è stato interesse a sviluppare modelli per la previsione della correttezza basati su *self-attention* oppure capaci di utilizzare anche le informazioni presenti nel testo degli esercizi. Queste due aree sono ancora poco esplorate per i risultati che promettono. Innanzitutto, proponiamo il modello Prediction Oriented Self-attentive knowledge Tracing (POST), composto da tre componenti: un *encoder* responsabile di modellare il contenuto degli esercizi passati, un decoder che aggiunge ad esso le informazioni sulle risposte passate e un secondo decoder che confronta queste informazioni con il contenuto del problema da prevedere. Successivamente, abbiamo utilizzato sei metodi (*CountVectorizer*, *word2vec*, *doc2vec*, *DistilBERT*, *Sentence Transformer* e *BERTopic*) per creare, a partire dai testi degli esercizi, dei vettori che li rappresentassero e abbiamo sviluppato delle versioni "NLP-enhanced" di DKT e POST capaci di utilizzarli. Abbiamo anche studiato come sviluppare modelli "ibridi", cioè capaci di usare contemporaneamente vettori provenienti da metodi diversi, e presentato due approcci per realizzarli a partire da "NLP-enhanced" DKT. La valutazione dei modelli è stata effettuata su tre dataset pubblici (*ASSISTments 2009*, *ASSISTments 2012* e *Peking Online Judge*) e uno privato fornito da *Cloud Academy*. POST migliora le prestazioni rispetto a SAINT+ e DKT su tutti i dataset valutati, con un incremento massimo rispettivamente del 6.65% per la binary accuracy e del 7.74% per l'AUC. "NLP-enhanced" DKT porta miglioramenti sui tre dataset pubblici, mentre "NLP-enhanced" POST supera la versione che non usa il testo su tutti i datasets, con un incremento più significativo su quelli con pochi dati disponibili. Infine, i modelli ibridi migliorano ulteriormente le prestazioni, affermandosi come miglior modello per il dataset *ASSISTments 2009*, con un incremento dell'AUC rispetto a DKT del 8.73%.

Parole chiave: Knowledge Tracing; elaborazione del linguaggio naturale; apprendimento profondo; self-attention; Transformer; previsione delle risposte

Contents

List of Figures

List of Tables

1 | Introduction

In the educational domain, Knowledge Tracing (KT) is the task of modelling the knowledge of a student over time, aiming at understanding its ability levels on different subjects. Being able to understand and describe students' abilities, starting from their observed performances on assessments and inferring unobservable traits, enables us to keep track of their level, personalize the learning process and enrich the experience on online education. Understanding the way students' knowledge improves over time can enable to control the learning process and optimize it. KT can be useful in online assessments to recommend particular questions, directly targeting students' weaknesses, or, for example, to guarantee more specific requirements, such as review and explore (i.e. suggesting users exercises about non-mastered concepts and new knowledge at the same time), smoothness of question difficulty (i.e. avoiding dramatic variations of the difficulty levels of exercises) and student engagement (i.e. being able to mantain student enthusiasm) [10].

Most approaches to KT rely on the assumption of having at least an ability (or "skill") associated with a question, which provides context information and can be used to model the knowledge representation. Without knowing an associated skill, logistic regression-based and probabilistic approaches must assume all items as related to a single skill to work, while others, such as Recurrent Neural Networks (RNN), memory-based networks or networks with attention mechanism, simply have worse performance.

Recent interest in KT has been in developing models able to understand relationships between questions directly from the textual content. Extracting the important content from text and representing it in a machine-friendly format can theoretically improve the performance. It could help understand relations between items and skills and enable the KT models to work on embeddings created from heterogeneous sources (assessments, small documents to read or even lesson transcripts).

Recently, some works in KT propose word frequency measures to compute similarities between texts, while others focus on generating an embedding for each text, using neural networks or Transformers. Each of them successfully preprocess text applying a Natural Language Processing (NLP) technique and passing the output as input to a KT

model. Comparing their results with models not using texts, they show improvements in the performance on the correctness predictions task, which is the task of predicting the correctness of students' answers to the next question at each time step.

To the best of our knowledge, no research has yet compared different techniques. Thus, we focus on performing a methodical analysis of some of the available NLP techniques, their advantages and disadvantages for the correctness prediction task.

First, we evaluate existing KT models, specifically focusing on the ones performing best without the text on the correctness prediction task. We chose Deep Knowledge Tracing (DKT) [19] for its relevance in the field and versatility and Separated Self-AttentIve Neural knowledge Tracing Plus (SAINT+) [25] for being the state of the art model in Ednet dataset [4], which is, to our knowledge, the largest publicly available educational dataset in terms of the total number of students, interactions, and interaction types.

Starting from DKT, we modify the architecture to receive as input an exercise embedding, as shown in Exercise Enhanced Recurrent Neural Network (EERNN) [13] and evaluate it using different NLP techniques to produce the exercise embeddings. We call this architecture NLP-enhanced DKT (NLP-DKT). We also propose two new models: Prediction Oriented Self-attentive knowledge Tracing with Multiplication (POST-M) and Prediction Oriented Self-attentive knowledge Tracing (POST), based on self-attention and composed of a past exercise content encoder, a past performance decoder and a prediction oriented module.

Then we extend Prediction Oriented Self-attentive knowledge Tracing with the ability to use as input exercise embeddings created using the different NLP methods. We denote this model as NLP-enhanced Prediction Oriented Self-attentive knowledge Tracing.

In the end, we show the possibility of combining different exercise embeddings to create hybrid approaches for NLP-DKT. We propose two examples of how to parallelize two or more NLP-enhanced DKT models, each using a different NLP method. Instead, for NLP-POST we suggest using a simple and effective method: summing together NLP generated embeddings with other embeddings (as skill embedding). We implement only the hybrid model for NLP-DKT.

We evaluate the proposed models on three public datasets: ASSISTments 2009[1], ASSISTments 2012[2] and Peking Online Judgement (POJ)[3] and on a private dataset from Cloud

[1] `https://sites.google.com/site/assistmentsdata/home/2009-2010-assistment-data/skill-builder-data-2009-2010`

[2] `https://sites.google.com/site/assistmentsdata/datasets/2012-13-school-data-with-affect`

[3] http://poj.org/

Academy (CA)[4], minimizing the Binary Cross Entropy loss and using as evaluation metrics the Binary Accuracy and the Area Under the Curve. The public datasets are stored and publicly available[5] for future research.

The contributions of this work can be summarised in the following points:

- We propose two self-attention based models: POST-M and POST, outperforming baselines using as information the exercise ids, the content ids (or skill ids), the correctness of previous answers and elapsed time between answers.
- We propose NLP-DKT and NLP-POST, extending respectively DKT and POST to use textual exercise embeddings as input ("NLP-enhancing" these models).
- We compare the utility of six NLP techniques (i.e. CountVectorizer, word2vec, doc2vec, DistilBERT, Sentence Transformer and BERTopic) to produce textual exercise embeddings to be used as input for NLP-DKT and NLP-POST.
- We suggest and evaluate two hybrid approaches to use at the same time exercise embeddings produced by different NLP methods for NLP-DKT. We suggest but not evaluate an approach for NLP-POST.

The code is publicly available[6] for future research.

The document is structured as follows:

- **Chapter 1** Introduction, shortly defining the topics of our work, their relevance and our contributions.
- **Chapter 2** Related works, where we give an initial description of KT task and the models relevant to our work.
- **Chapter 3** Background, providing a theoretical explanation of all the terms needed to fully understand our work, including deep learning, attention mechanism and NLP techniques.
- **Chapter 4** Models, where we accurately perform an analysis of previous relevant KT models, their architectures and their importance in our work.
- **Chapter 5** Proposed models, where we show and explain the models we develop, their advantages and how to enhance them with NLP exercise embeddings.

[4]https://cloudacademy.com/
[5]https://drive.google.com/drive/folders/1rON8zS9oPvIxo9QZTRu_8MkY3ueCpe5n?usp=sharing
[6]https://github.com/SimoneSartoni/POST—NLP-for-KT

- **Chapter 6** Experimental setups, describing the four datasets we use for evaluation and the data processing, split and batching needed to use them. In the same chapter we describe the loss, metrics and hyper-parameters we choose for training and evaluation.
- **Chapter 7** Results, where we show the performance of each proposed model and the NLP methods to enhance them, comparing them with the baselines (DKT and SAINT+) and discussing them.
- **Chapter 8** Conclusions, where we summarize the results, in order to extract conclusions. Then we suggest some topics, which should be the focus for future works.

2 | Related Works

In this chapter, we describe Knowledge Tracing from a formal point of view, we introduce state of the art models for the correctness prediction task, some recent works which used NLP techniques to improve KT and their relevance in our work.

2.1. Knowledge Tracing

Knowledge Tracing (KT) is the task of modelling the evolution of the knowledge of students over time. In our work we consider KT focused on predicting the correctness of future students' answers.

Given a student S, we denote the act of submitting answer r_t to exercise question e_t at time t as an interaction $I_t = (e_t; r_t)$. We can describe KT as the task of estimating, given the past interactions from time $t = 0$ to time $t = T - 1$, the answer of the student at time $t = T$, denoted by r_T, to exercise e_T. Formally, from (e_0, r_0), (e_1, r_1), ... (e_{T-1}, r_{T-1}) and e_T , KT is the task of predicting value r_T. Many models assume to have a skill s_t associated with each question e_t, which can represent the main ability needed to answer correctly, or possibly a category or label. Knowing the skill enables an easier representation of knowledge, but at the same time the performance of the models based on this assumption could be limited by its availability (not guaranteed on many datasets).

2.2. Bayesian Knowledge Tracing

The first model we consider is Bayesian Knowledge Tracing (BKT) [6], which describes each skill as a binary variable and makes use of a Hidden Markov Model, based on four parameters per skill representing the probabilities of:

- knowing the skill;
- learning the skill after interacting with an exercise related to it;
- guessing the answer despite not knowing the skill;

- slipping, meaning answering not correctly, despite knowing the skill.

This model tries to mimic directly the human behaviour but has shown some limitations [19] due to the binary variable representation of skills, which may be unrealistic.

2.3. Deep Knowledge Tracing

In 2015 Deep Knowledge Tracing (DKT) [19] showed the capabilities of Recurrent Neural Networks (RNN) [22] in modelling this task, using a hidden dimension to represent and track latent knowledge over time and enabling to reuse past information later in time. In DKT skills are not assumed as independent variables; thus, it can learn automatically more complex skills (as combinations of existing ones) and relations among them. If the skills associated with the items are unknown, BKT assumes all the questions assess the same skill, while DKT is able to understand multiple skills, directly learning hidden representations of items and relations between them.

DKT has outperformed BKT on most of the datasets, improving, for example, the Area Under the Curve metric on ASSISTments 2009 dataset from 0.68 to 0.85. It is able to mimic a function considering latent concepts, the difficulty of each exercise, the prior distributions of student knowledge and its evolution over time, while BKT degrades exponentially with the number of hidden concepts growing, not being able to learn unlabelled ones. However, DKT needs large amounts of data, being well suited for online education and not for small classrooms.

DKT has also been implemented using Long Short Term Memories (LSTM) [9] to develop a more powerful variant, able to learn some coefficients representing the amount of past information to maintain at each time step. We choose DKT implemented with LSTM as the first baseline to compare our models.

2.4. Attention based models

Recently, attention-based technologies have been shown to outperform deep neural network approaches in many tasks (e.g. sequence classification, language modelling and summarization) [29] and there have been several attempts to develop attention based architectures to model KT.

Self Attentive Knowledge Tracing (SAKT) [17] is the first model to apply self-attention to KT. SAKT uses exercise ids as knowledge concepts and learns which of the previous interactions are relevant for the next exercise, associating an higher weight to them. Then

SAKT predicts correctness using learnt weights as a weighted sum of the correctness of previous exercise answers. Compared to DKT, SAKT has shown some improvements on the Area Under the ROC Curve (AUC) metric, but the main advantage is the possibility of parallelizing computation, making the model orders of magnitude faster.

Separated Self-AttentIve Neural Knowledge Tracing (SAINT) [5] has instead introduced the Transformer model to KT, which is a more complex encoder-decoder architecture with self-attention layers as basic blocks. SAINT has been extended to use temporal features in the so-called SAINT+ [25] architecture, which is the state of the art model on Ednet dataset [4]. SAINT+ is the second baseline we compare our models with.

2.5. Natural Language Processing for KT

Generating additional information about the question from its text can lead to improvements in KT. For example, transforming raw text into useful fixed-length vectors and developing KT models working on generic vectors as input, can enable to use together information from heterogeneous sources, such as text, skill and temporal features. Different works focused on applying NLP techniques to create a useful embedding from the text of each exercise.

Exercise-Enhanced Recurrent Neural Network (EERNN) [26] produces an embedding for each word using word2vec [16] and uses a bidirectional LSTM network to learn relations between words and use last hidden state as embedding for the exercise. Then EERNN forwards these embeddings as input to another LSTM network responsible for tracing knowledge over time, as DKT. For final prediction, EERNN proposes to use target exercise embedding concatenated alternatively to the last hidden state of the previous network (EERNN with Markov Property) or to the output of an attention mechanism applied over its past hidden states (EERNN with Attention mechanism).

Exercise-aware Knowledge Tracing (EKT) [13] is an extension of EERNN by the same authors, where they move from a vector to a matrix representation of knowledge, where columns represent the different skills available. This change enables the creation of a model able to learn hidden representations from the text (as EERNN), returning at the same time a more explainable description of knowledge evolution over time provided by the output before the prediction layer.

Instead, Exercise Hierarchical Feature Enhanced Framework (EHFKT) [28] approaches the problem by using BERT [8] to create word vectors. Those embeddings are then passed to different systems to estimate information, hierarchically structured in three levels: the

difficulty, the skills and the semantic group. The extracted information is later used as input to an LSTM network (as EERNN) to model knowledge evolution over time.

Lastly, Relation-Aware Self-Attention for Knowledge Tracing (RKT) [18] is an extension of SAKT made by the same authors, preprocessing similarity coefficients between exercises, based on text similarity, on time between the interactions and on the performance of users. Those additional coefficients are summed to the outputs of the SAKT self-attention layer.

Looking at the results from [18], where authors compare DKT, SAKT, EERNN, EKT and RKT on three datasets, we can see how RKT is the best performing model on each dataset. It is also remarkable to see that all the models using textual information (EERNN, EKT and RKT) show better performance than DKT and SAKT. EHFKT has been shown to outperform both DKT and EKT on another dataset. These results confirm our idea that a methodical analysis of the NLP techniques available and how to use them can further improve KT. Understanding each NLP technique's advantages and disadvantages can enable a more aware choice of the most suitable one.

In our work, we follow the approach from EERNN to generate exercise embeddings from text and then develop a KT model using them as input. Similarly to EERNN, we propose an LSTM network able to focus on the target exercise embedding. In the end, we propose two new self-attention architectures able to focus more on target exercise embedding than SAINT+ and we extend them to receive as input NLP generated exercise embeddings.

3 | Background

In this chapter, we introduce and describe all the necessary notions to understand our work, to create a common theoretical ground. The main subjects are deep learning, attention mechanism and Natural Language Processing techniques.

3.1. Deep learning

Deep learning [7] is a subset of machine learning based on artificial neural networks and aiming at reproducing the information processing and the distributed communication nodes in biological systems. An artificial neural network is a system composed of different layers of parallel neurons, connected in series. To explain better artificial neural networks, we start from the concept of neurons.

3.1.1. Perceptron

A neuron is the basic unit of artificial neural networks and consists of a perceptron [21], a mathematical function taking N different inputs, each one, denoted by X_n, associated to a certain weight W_n and such that the neuron "activates" only when the weighted sum of inputs plus a bias b satisfies a certain activation function f. So the perceptron can be defined as a binary classifier that applies a threshold function f to a linear combination of the inputs. Function describing the perceptron is:

$$y(x) = f(W^T X + b) = f(\sum_{n=1}^{N} W_n * X_n + b) \tag{3.1}$$

3.1.2. Feed forward neural network

An artificial neural network (ANN) is an architecture composed of concatenated layers, where outputs of K_{th} layer are the inputs of $K + 1_{th}$; a layer is composed of many neurons in parallel, enabling fault tolerance and contemporaneously focusing on different aspects

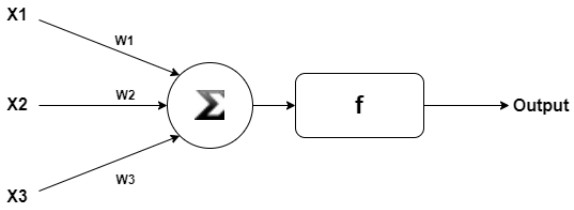

Figure 3.1: Perceptron with three inputs.

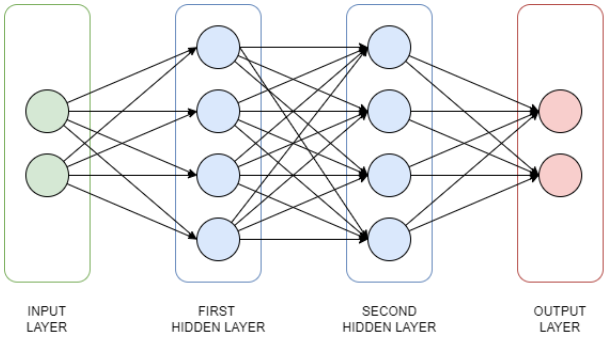

Figure 3.2: Feed forward neural network with two inputs in the input layer, two hidden layers, each with four neurons and an output layer with 2 outputs.

of inputs. The term "deep" comes from the ability of ANN to understand hierarchical and deep features from unstructured data, eventually needing only some preprocessing of the inputs. Once ANN receives inputs, they are passed through the layers, in the so-called "forward propagation" phase, producing the output of the model.

To improve performance and learn the target function, during the training phase outputs are used to compute a chosen loss function, which represents the error in the results and is used to change the weights of neurons starting from the output in the opposite direction of the previous step. This operation is called **back-propagation** and updates are performed applying these formulas over each neuron at each time step:

$$W_i^{k+1} = W_i^k + \nabla W_i \tag{3.2}$$

$$\nabla W_i^k = \eta * t^k * X_i^k \tag{3.3}$$

where W_i^k is previous weight at time step K for input i, W_i^{k+1} is updated weights (at time step $K+1$) for input i, η is learning rate and t^k is desired output to achieve for that sample K of experience.

The first type of ANN developed has been Feed Forward Neural Networks (FFNN), where layers are concatenated one after the previous, making the flow of execution moving only

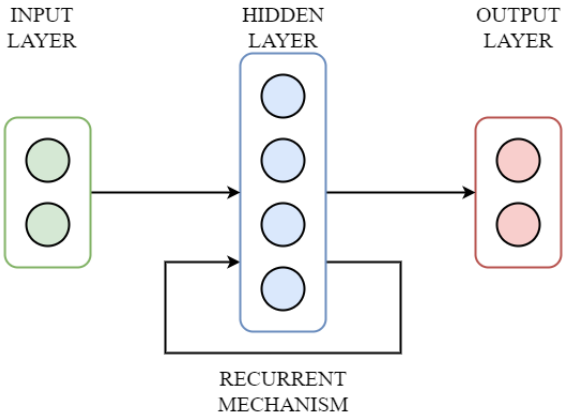

Figure 3.3: Recurrent neural network example with two inputs, four hidden units and two outputs.

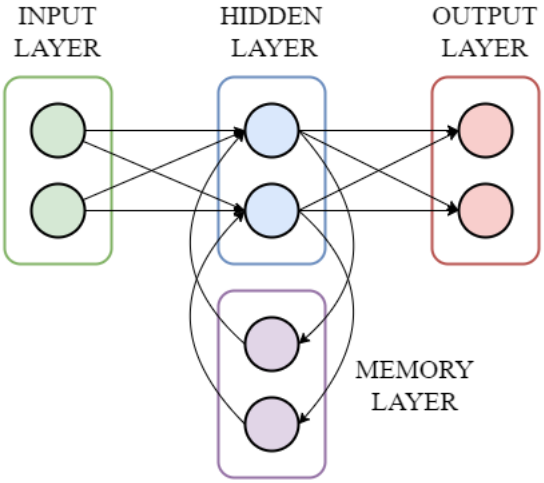

Figure 3.4: Elman Recurrent Neural Network example with two units for each layer: input, hidden, memory and output.

towards results, without cycles or loops, enabling easy computation of gradient and easy back-propagation. We define a Feed-Forward layer as an FFNN with a single hidden layer; we'll reuse this term multiple times in this document.

3.1.3. Recurrent Neural Network

Recurrent Neural Networks are artificial neural networks where there is at least a layer receiving as input some output from another layer (or itself) from previous time-steps. For example, a layer can receive as inputs at time step $K+1$ input X_{k+1} (as FFNN) and its output at time step K, h_k, allowing to maintain information from past time steps to be reused in the future. A variant of basic RNN, called Elman RNN, uses an additional hidden layer to model memory, highlighting better the difference between FFNN and RNN. RNN and Elman RNN are shown in Figures 3.3 and 3.4 respectively.

The method to optimize RNN to learn target function is called Back-propagation Through Time (BTT), adding an unfolding phase before computing back-propagation in FFNN. This architecture is used to model temporal dynamic behaviours or sequences of repeated data, but suffers from two main problems due to BTT: "vanishing" gradient (tending to zero, due to $lim_{n->\infty}W^n = 0$) or "exploding" (tending to infinity, due to $lim_{n->\infty}W^n = \infty$) gradient, which makes difficult to learn long dependencies.

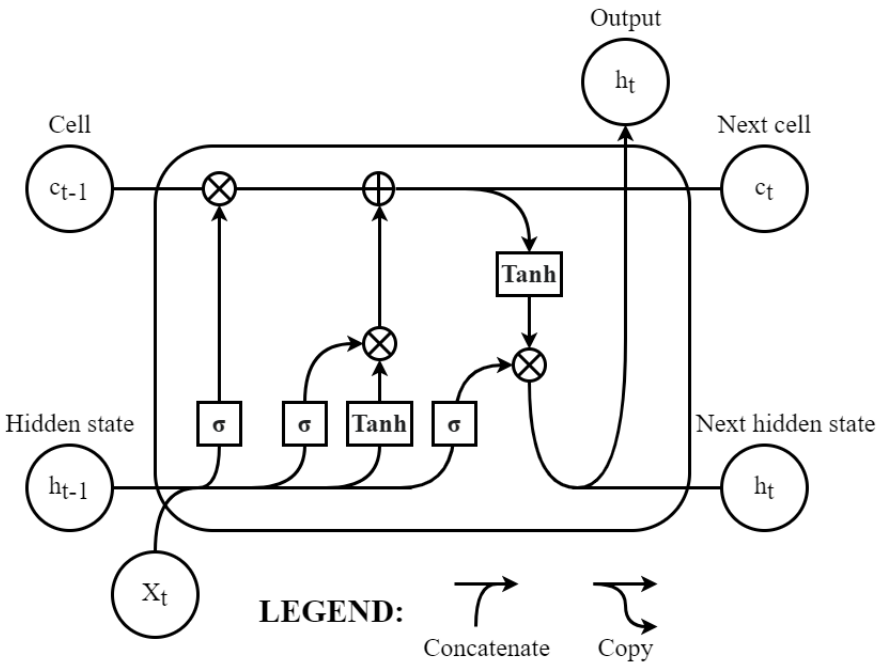

Figure 3.5: Long Short Term Memory unit.

3.1.4. Long Short Term Memory

Long Short Term Memory (LSTM) [9] network is a variant of Recurrent Neural Network, which replaces the recurrent mechanism with a memory-based mechanism, relying on LSTM units, which are neurons able to decide importance to give to current input and previous past. An LSTM unit consists of a function with four vectors: an input/update gate vector, a forget gate vector, a hidden state vector (maintaining value over time) and an output gate. Those gate vectors are applied according to following equations in the forward pass:

$$f_t = \sigma_g * (W_f * X_t + U_f * h_{t-1} + b_f) \tag{3.4}$$

$$i_t = \sigma_g * (W_i * X_t + U_i * h_{t-1} + b_i) \tag{3.5}$$

$$o_t = \sigma_g * (W_o * X_t + U_o * h_{t-1} + b_o) \tag{3.6}$$

$$\tilde{c}_t = \sigma_c * (W_c * X_t + U_c * h_{t-1} + b_c) \tag{3.7}$$

$$c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t \tag{3.8}$$

$$h_t = o_t \circ \sigma_h(c_t) \tag{3.9}$$

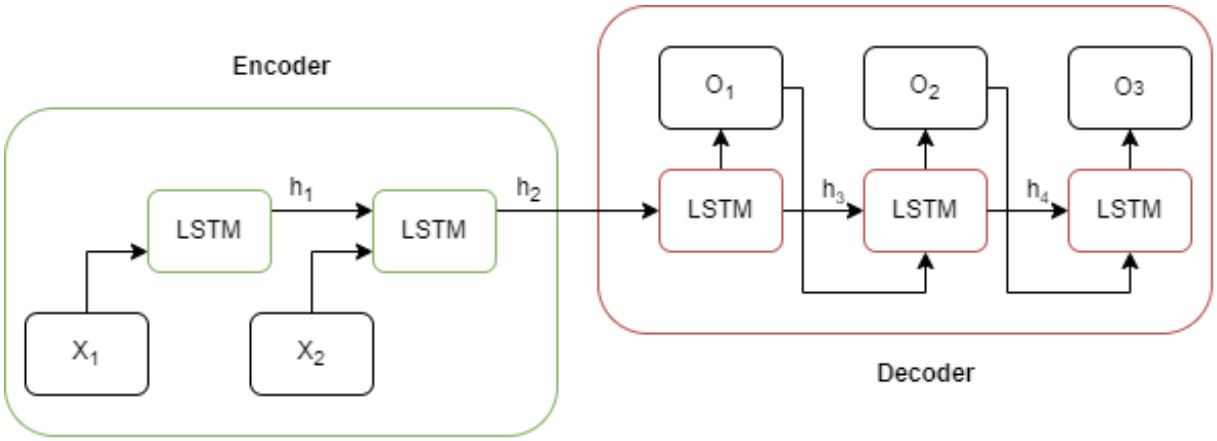

Figure 3.6: Encoder Decoder architecture for Seq2Seq problem using LSTM units.

where, at time t, x_t is the input vector with dimension d, f_t is forget gate activation vector, c_t is memory cell input activation vector , o_t is output gate activation vector, and h_t is the final output of the unit. σ_c is the sigmoid activation function, while σ_h is the hyperbolic tangent function.

LSTM units solve the vanishing gradient problem, enabling the network to decide and optimize the importance given to the past. LSTM networks still have two main limitations: the difficulty of focusing on inputs too far in the past and the impossibility of parallelizing computation, because at each time step $t + 1$ it needs the output of time step t as input.

3.2. Encoder Decoder Sequence to Sequence

Encoder-Decoder model (Seq2Seq) [27] has been introduced to solve the problem of mapping a fixed-length input to a different fixed output length, but is useful in many other contexts too. It is a neural network composed of three main components: an encoder, an intermediate vector and a decoder.

The encoder is a sequence of LSTM (or recurrent) units over fixed input length, where each unit accepts a single element of the input sequence and the hidden state of the previous unit; encoder outputs is the hidden state generated by the last LSTM unit, which is used as an intermediate vector. Instead, the decoder is a sequence of units, each one accepting as input the hidden state of the previous unit and its output and producing as output an element of the output sequence. The first unit of the decoder accepts the final hidden state from the encoder. The idea of using two LSTM networks (one for encoder, one for decoder) enables the Encoder-Decoder model to address many different problems, removing the constraint of the same length of input and output sequences.

3.3. Attention

Attention is a mechanism first introduced by Bahdanau et al. (2014) [1] to solve the limitations of the encoder-decoder model for the Seq2Seq task, in particular the weak availability of data from the encoder to the decoder. The decoder receives as input only the last hidden state of the encoder, which must summarize all the relevant information about each element of the input sequence. This sequential approach is the bottleneck of the encoder-decoder system. Attention mechanism mimics the cognitive attention, which is the ability to focus on discrete subparts of information available; it was born to solve seq2seq problem but has been extended to other tasks too, being useful to model temporal or spatial dimensions, different features or different elements of memory.

Attention has shown to be successful in many fields, such as NLP and computer vision, leading to the development of state of the art models. Different forms of attention have been developed, such as dot product, query-key-value, Bahdanau [1], or Luong [14]; initially, they were applied to models with RNN, keeping the limitation of not being parallelizable. We will focus on more recent Multi-Head self-attention, which the Transformer [29] model is based on, due to the absence of the non-parallelizable limitation and the consequent better performance it offers.

3.3.1. Multi-Head self-attention

An attention function can be described in general as a mapping between a query, Q, and a sequence of key-value pairs (K, V), where all inputs and output are vectors. Transformers use Scaled Dot Product [29] as attention, which consists of the function:

$$ScaledDotProductAttention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{3.10}$$

Scaled Dot Product attention is a variant of Dot product attention using $1/d_k$ scale factor, where d_k is the dimension of query, key and value vector. Multi-Head attention is finally computed by multiplying Q, K, V for different parameter matrices W_i^Q, W_i^K, W_i^V and applying scaled dot product on the newly obtained queries, keys and values:

$$MultiHead(Q, K, V) = Concat(head_1, ..., head_h)W^O \tag{3.11}$$

where

$$head_i = ScaledDotProductAttention(QW_i^Q, KW_i^K, VW_i^V) \tag{3.12}$$

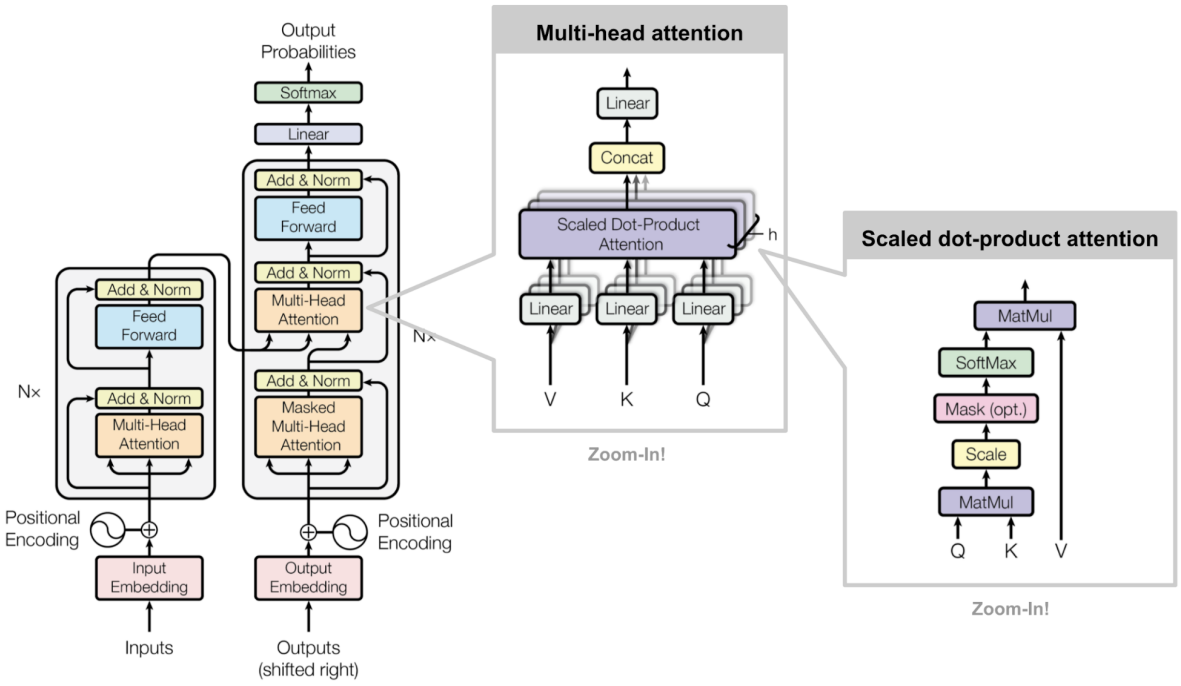

Figure 3.7: Representation of the architecture of the Transformer model [29]. The image is taken from https://lilianweng.github.io/posts/2018-06-24-attention/ and refers to Figures 1 and 2 from the original paper [29].

3.3.2. Transformer

The Transformer is a recent deep learning model proposed in the paper *Attention Is All You Need* [29], completely based on self-attention to draw global dependencies between inputs and outputs. This model is divided into 2 components: encoder and decoder, each one being a complex architecture itself.

The encoder is a stack of N identical layers, each one composed of 2 sub-layers: a multi-head self-attention mechanism and a position-wise fully connected FFNN; each sub-layer adopts residual connection (summing input to output) and applies normalization to output vectors. The decoder is similar to the encoder but adds each layer an initial sub-layer, responsible for applying masked multi-head attention, avoiding attending to information from outputs not already available at the considered element of the target sequence. The encoder is responsible for generating N (number of heads) sequences of vectors, representing information encoded from the input sequence at each step, which is then given as Keys and Values to the decoder, while queries will be the outputs of the previous steps (shifted by 1 in the past/to the right).

3.4. Natural Language Processing

Natural Language Processing is defined as the branch of Artificial Intelligence responsible for enabling computers the ability to understand and interact with the human language (both in written and spoken form), in a similar way to how human beings can. Another possible definition [12] describes NLP as a theoretically motivated range of computational techniques for representing natural texts at one or more levels of syntactical or semantical analysis to reach human-like language processing for different tasks or applications. Understanding natural language is a very complex task, due to the presence of complex sentence structures, ambiguities, homonyms, homophones, sarcasm, idioms and many other difficulties. However, it has already influenced most of our life with the development of automatic translators, chatbots, intelligent assistants, spam filters and summarization tools. A lot of tasks have developed during the last decades and have been improved thanks to rule-based, statistical, machine learning, and deep learning techniques. In particular, we'll describe the techniques useful to capture the meaning and the features present in a text and encode them in a text embedding, to be used for other tasks.

3.4.1. Text cleaning process

To efficiently use the text of the questions to generate useful information, we first need to process the text, dealing with the most common related problems. Those processing actions are usually referred to as text cleaning techniques. In our work the most useful ones have been the following:

Remove HTML tags

Since KT's main goal is to support online learning, we usually deal with text derived from HTML web pages, with consequent HTML tags, making the text difficult to comprehend for computers. So we need to remove tags, translating for example sentence " Does Amazon S3 provide a filesystem? " in "Does Amazon S3 provide a filesystem".

Remove stopwords

In any language, some words appear too often to be useful and do not provide any additional information about the content. Those words are called stopwords. In addition, if our texts are topic-specific, many words, generally not considered stopwords, can become unuseful and are called document-specific stopwords. A simple example in a mathematical

only context can be the word "number", present in most of the texts, becoming unuseful. Some NLP semantic embedding techniques autonomously deal with stopwords, while others need stopwords to be removed. There are many libraries available to remove common English stopwords (such as *a*, *and*, *then*, etc), while removing document-specific stopwords requires to use statistical techniques, related to word frequencies.

Stemming and Lemmatization

Stemming is the process of removing the last characters from words, to reduce plurals, derivations, etc to the basic form. Lemmatization is an improvement of stemming, which consists of the process of obtaining from a word its meaningful basic form, also considering the context. They are alternative options to reduce the number of possible words and understand easier their meaning and context. According to [2], lemmatization has been shown to perform better in the document retrieval task, so we choose this method to preprocess our texts, instead of stemming. As stopwords removal, lemmatization is not needed for some NLP semantic embedding techniques, while for others (as approaches computing word frequencies) it is needed to improve quality.

Tokenization

Tokenization is the process of decomposing a text (represented by a string) into a sequence of words and then associating to each word a unique integer identifier, allowing computer systems to easily manage the text as a sequence of identifiers. Tokenization is essential for any NLP semantic embedding technique.

3.4.2. CountVectorizer

The simplest approach to represent a text in a machine-friendly format is CountVectorizer, consisting of:

- collecting all the words which appear in our set of texts, counting and ordering them.
- Each text will be described by a vector with a dimension equal to the number of words, whose element at position K will be equal to the number of times the word associated to that position appears in the text.

CountVectorizer is sometimes referred to as Bag Of Words (BOW) too.

3.4.3. TF-IDF

Term Frequency-Inverse Document Frequency (TF-IDF) is a statistical method to evaluate the relevancy of words in a set of documents. It consists of computing for each word in each document two metrics: Term Frequency, representing the frequency of the word in the considered document, and Inverse Document Frequency, representing the importance of the word over all the texts. Those values are computed as:

$$TF(doc, word) = \frac{\text{Number of occurrences of word in document}}{\text{Number of words in text}} \tag{3.13}$$

$$IDF(word) = \log(\frac{\text{Number of documents}}{\text{Number of occurrences of word in the document}}) \tag{3.14}$$

$$\text{TF-IDF(doc, word)} = \text{TF(doc, word)} * \text{IDF(word)} \tag{3.15}$$

TF-IDF is used to improve representation provided by CountVectorizer and is usually considered a better model.

3.4.4. Word embedding approaches

Creating word embeddings is a different approach to NLP in which a model can associate to each word a unique word embedding, which is a vector of continuous values with a fixed size. Word embeddings encode both syntactical and semantical relations between words. Different models can produce word embeddings, we consider word2vec (with Continuous Bag Of Words or Skip-Gram architectures) in Section 3.4.4 and both Bidirectional Encoder Representations from Transformers (BERT) and DistilBERT in Section 3.4.4, due to their relevance and performance on NLP tasks.

Word2vec

Word2vec is one of the most popular techniques for NLP, introduced in 2013 [15], and consists of using a simple neural network with an input, a hidden and an output layer. Each word is initially represented by a one-hot encoding (equal to BOW representation of a text with only that word).

Embeddings can be created by using two different approaches: Continuous Bag Of Words (CBOW) [15] and Skip-Gram [15]. CBOW network takes as input a context of C words one-hot encodings (the C words before the target word) and tries to predict next word one-hot encoding as output. While Skip-Gram takes as input the target word encoding and outputs C vectors of probabilities, which should represent the probability. In both

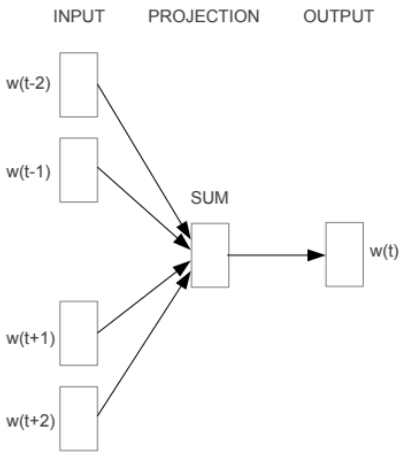

Figure 3.8: CBOW architecture. The image is taken from the original paper [15].

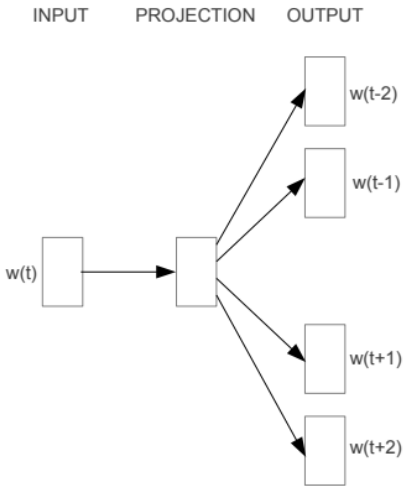

Figure 3.9: Skip-Gram architecture. The image is taken from the original paper [15].

models, the hidden state is used as word embedding (allowing reducing the dimension by orders) and the parameter C is called window size. CBOW usually is faster to train than Skip-Gram, while the second tends to provide better results and is able to create accurate embeddings of rare words too.

BERT and DistilBERT

BERT approaches the same task of creating syntactic and semantic aware word embeddings and introduces a completely new architecture based on transformers. BERT was developed by the Google AI language team in 2018 [8] and became the state of the art on eleven NLP related tasks. BERT is much important because it is a very generic and effective architecture, pre-trained on two generic tasks, able to be easily extended for other tasks without the need to retrain the model. BERT architecture is a multi-layered Bidirectional Transformer pre-trained on two different tasks in order:

1. Masked Language Modeling (MLM), which consists of giving as input to the model a sentence with some random words "masked" and training the model to infer those words from the unmasked part of the sentence. Being able to infer the words means the ability to learn the context meaning and the syntactical structure of the sentence.

2. Next Sentence Prediction (NSP) task has as inputs two sentences: A and B, where B is half of the times the sentence following A in the source text and the other half an uncorrelated random sentence from another text. The transformer model is

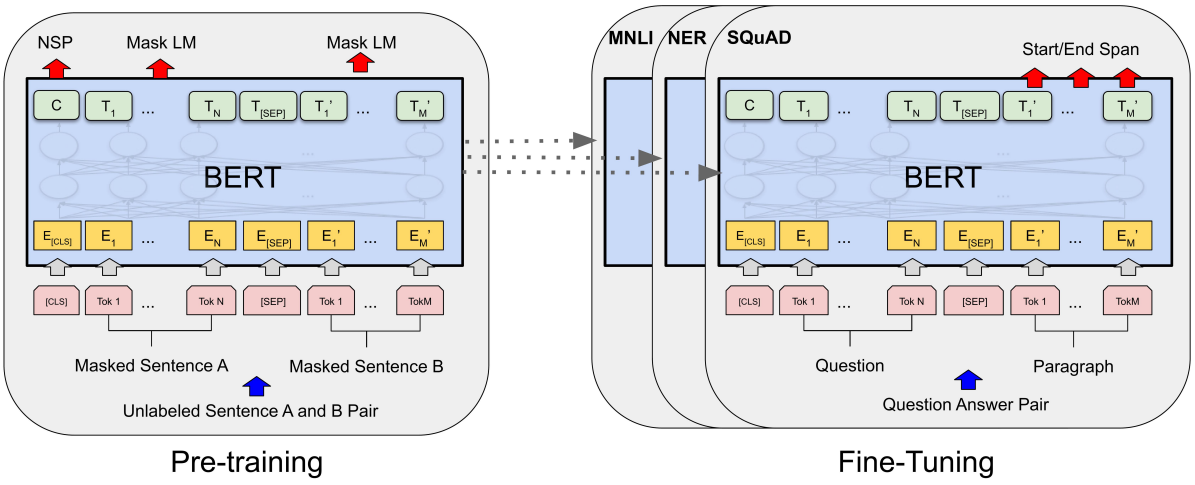

Figure 3.10: BERT model pretraining and fine-tuning.

trained to predict whether B is the "next" sentence of A, or not.

The importance of BERT is the possibility to extend the model for many other tasks requiring textual embeddings, by fine-tuning the embeddings produced at the end of the pretraining phase. Fine-tuning for another task T consists of adding some additional layers at the end of the network to produce the target outputs and train the whole architecture to understand a function for T. Fine-tuning makes the model focus only on the target function for the specific task and avoids the burden of recomputing semantic and syntactical information, which is computationally and memory expensive.

DistilBERT [23] is a more compact version of BERT obtained using distillation [3], whose performance are similar to BERT (nearly 97%), but having 40% fewer parameters and being 60% faster at inference time. Distillation is a technique to train a student model to mimic the function of another model, used in practice to develop smaller student models with performance similar to the more complex (and consequently memory expensive) teacher model.

3.4.5. From word to document or sentence embedding

Generating word embeddings with the previously shown method is a very interesting starting point for NLP related tasks, in particular for translator applications, chatbots or document retrieval. However, word embeddings are very difficult to be used as direct inputs to KT models, due to the memory and computational power required. Having an embedding (with size d_e) for each of the n_w words in a sentence, means we can associate to each sentence a textual embedding with size $d_e * n_w$. For many tasks this embedding size is too large, ending up being out of memory, so recent approaches have focused on

developing ways to compute sentence embeddings with fixed size d_e.

We decide to focus mainly on some of the existing approaches: Pooling, doc2vec and SentenceTransformers.

Average, Max

The simplest approach to obtain sentence/paragraph embeddings from word embeddings is to apply a reduce operation over the words dimension: $O : (R^{n_w}, R^{d_e}) -> R^{d_e}$. The common operations to apply are average or max over words dimension. Average of words embeddings has shown to be useful to refine word embeddings for other tasks and is still considered a viable and easy option.

Doc2vec

Doc2vec [11] is an approach to extend word2vec architecture to be trained at the same time to produce a sentence/paragraph embedding. It is a model capable to produce fixed-length embeddings from a sequence of words with variable length (as sentences, paragraphs or documents). In addition to the shared matrix W, whose rows represent word embeddings, doc2vec (also called Paragraph Vector) maintains a matrix D, whose row D_i represents embedding for sentence/paragraph i.

Two variants have been proposed: Distributed Memory Model of Paragraph Vectors (PV-DM) and Distributed Bag of Words version of Paragraph Vector (PV-DBOW). PV-DM consists of adding to inputs of CBOW model [16] an initial embedding of the paragraph, computed from matrix D, shared across all training samples of that paragraph and learnt during training. PV-DBOW architecture instead trains the model to predict the words in a certain window, given the paragraph ID. According to [11], PV-DM alone usually works well for most tasks, but its combination with PV-DBOW is usually more consistent across many tasks and therefore is recommended.

Sentence-BERT and Sentence Transformers

Sentence-BERT (SBERT) is a model proposed by Nils Reimers and Iryna Gurevych in 2019 [20] and consists of an extension of BERT to enable embeddings to describe efficiently sentence meaning and similarity. As described by the authors, previous ideas of using BERT for sentence description consisted of averaging BERT embeddings, but this method performs worse than computing the average of embeddings from other models, such as word2vec.

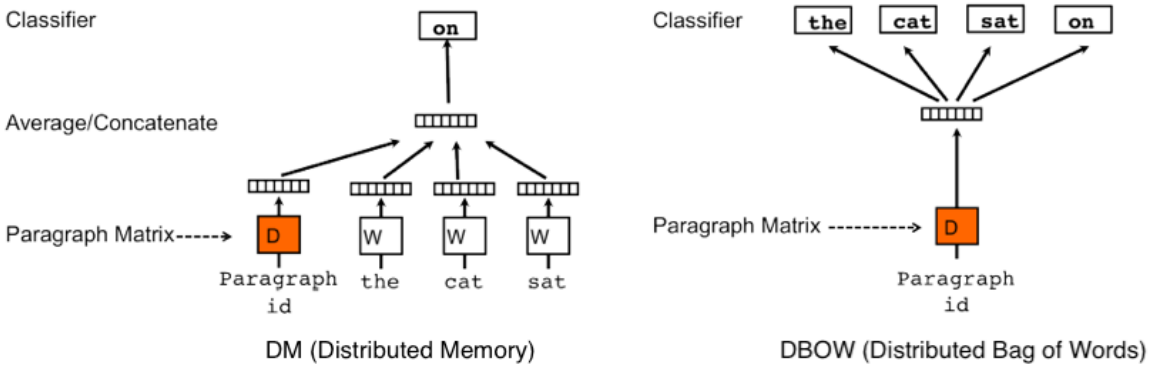

Figure 3.11: PV-DM and PV-DBOW models respectively.

SBERT adds a Pooling layer to generate a single fixed-length embedding from words embeddings of each sentence, by computing an operation (AVG, MAX or CLS); then it fine-tunes the whole network (BERT, Pooling, Normalization and softmax classifier) to produce sentence embeddings which optimize two tasks of sentence similarity, using both Siamese and triplet networks [24] and cosine similarity.

Another advantage of SBERT is the ability to infer semantic similarity by simply pooling words embeddings and computing cosine similarity, without the need to use parameters to learn similarity scores between each pair of input sentences. This allows the model to be computationally efficient.

Sentence-Transformers[1] is a library implementing Siamese network on top of different transformers (such as BERT, DistilBERT, *all-mpnet-base-v2*, etc.). We use this library because it provides a lot of already implemented models. According to the published performance about sentence embeddings and semantic search, the best performing model is the Siamese network on top of *all-mpnet-base-v2* model, so use it as the fifth NLP method. Henceforth, we refer to this model as "Sentence Transformer".

3.4.6. Topic prediction

Topic prediction is a different task from word or sentence embeddings creation, which consists of automatically understanding from a set of texts some relevant topics, grouping texts according to them and estimating probabilities of each document belonging to a topic.

[1] https://www.sbert.net/docs/pretrained_models.html

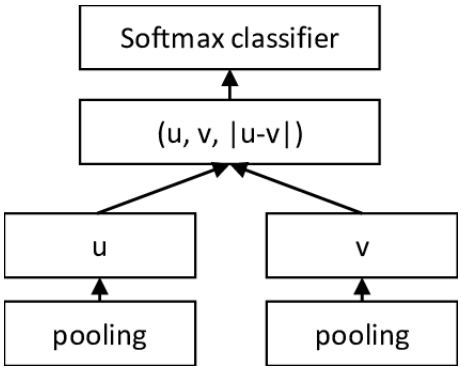

Figure 3.12: Sentence BERT fine-tuning architecture.

BERTopic

We consider a particular method, called BERTopic[2], proposed by Maarten Grootendorst in 2020 on its Github page, consisting of using sentence embeddings as input to a model which first groups document into semantically similar clusters and then creates topic representations. We do not enter into the details of the method, because we only want to use it to see if a topic-based approach can add some value compared to directly using sentence embeddings.

Estimated topics are more human-friendly and similar to skills, so we want to see if adding additional clustering operations (mainly unsupervised) after sentence embeddings generation improves or reduces performance. However, this is not the main subject of our work, so we won't focus on optimizing this method.

[2] https://maartengr.github.io/BERTopic/

4 | Previous models in Knowledge Tracing

In this chapter, we show in detail the models already presented in Chapter 2, providing a complete description of their architecture, the input data they make use of, their performance and their relevance in our work. The models we describe are Deep Knowledge Tracing in Section 4.1, Self Attentive Knowledge Tracing in Section 4.2, Relation-Aware Self-Attention Knowledge Tracing in Section 4.3, Separated Self-AttentIve Neural Knowledge Tracing in Section 4.4, SAINT+ in Section 4.5 and Exercise-Enhanced Recurrent Neural Network in Section 4.6. We do not enter into the details of EKT and EHFKT, considering that their only relevance to our work is the use of BERT to produce exercise embeddings taken from EHFKT.

4.1. Deep Knowledge Tracing

Deep Knowledge Tracing [19] is the first model in KT to use deep neural networks to describe and track the knowledge of students over time. Usually the input to the model is the one hot encoding of value $x_t = 2 * e_t + r_t$. The first version of DKT model (denoted as "Vanilla") consists of a Recurrent Neural Network with 3 layers:

- an input layer with dimension $D = 2 * d_{item}$, where d_{item} is the number of unique item ids.
- an hidden layer with recurrent units
- an output layer, producing a vector with dimension D, each one representing the probability of answering correctly to next question e_{t+1}.

If skills associated to items are available, we can decide to track their evolution over time. In this case the input is the one-hot encoding of value $x_t = 2 * s_t + r_t$, the dimension is equal to $D = 2 * d_{skill}$, where d_{skill} is the number of unique skill ids. DKT architecture is shown in Figure 4.1.

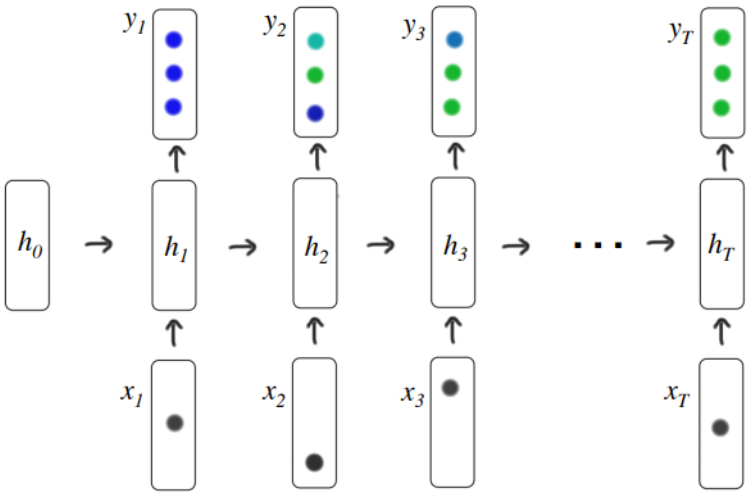

Figure 4.1: DKT architecture, where inputs X_t are the one-hot encodings of question ids and prediction y_t is a vector representing the probability of getting each of the dataset exercises correct. The intermediate layer is a RNN. The image is taken from the original paper [19].

4.1.1. Long Short Term Memory variant

A variant of Vanilla DKT has been proposed in [19] to exploit the advantages of LSTM networks. The LSTM variant has been shown to perform better, so we consider it as the first baseline. The baseline will be computed as the best performing between using item ids or skill ids (when available) for the one-hot encoding.

4.2. Self Attentive Knowledge Tracing

To the best of our knowledge, Self Attentive Knowledge Tracing (SAKT) [17] has been the first model to use self-attention to overcome the limitations of DKT, enabling the model to decide at each time step how much importance to give to each of the previous interactions.

As DKT, the model is able to predict answer r_{t+1}, given as input the interactions $x_0, x_1, ..., x_t$ and the exercise question to answer at next time step e_{t+1}, where $x_i = (e_i, r_i)$. SAKT consists of three sub-modules: an embedding layer, a self-attention layer and a feed-forward layer with sigmoid activation function on top to compute predictions.

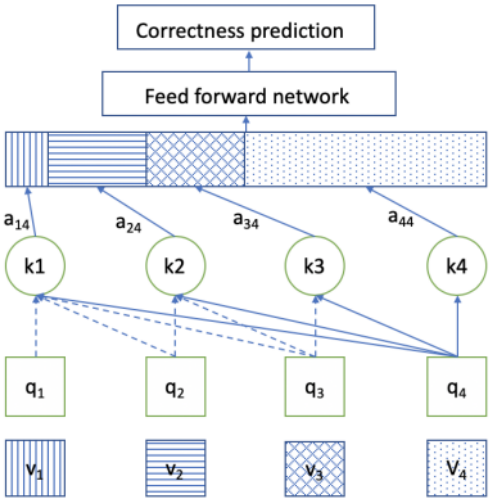

Figure 4.2: SAKT architecture, consisting of a self-attention layer estimating at each time-step weights only for each of the previous elements. The image is taken from the original paper [17].

4.2.1. Embedding layer

Denoted E the number of different exercises, input interactions are preprocessed into sequences of a single value $y_0, ..., y_t$, where $y_i = e_i + r_i * E$. This embedding function can seem different from DKT one, but in practice, it has the same role: it associates with each combination of exercise and answer a unique value. SAKT can receive as input only sequences with constant length N, so if $t < N$ sequence is padded into $y_0, ..., y_N$, while if $t > N$ it is divided into subsequences with length N and the last one is padded, if necessary.

Then, values are embedded using an interaction embedding matrix $\mathbf{M} \in R^{2E \times d}$ into a vector with dimension d, obtaining sequences of embeddings $M_{y_o}, ..., M_{y_N}$. A similar procedure is done to embed exercise e_{t+1} with matrix $\mathbf{E} \in R^{E \times d}$. Finally, embeddings are summed to a positional embedding, created using positional embedding matrix $\mathbf{P} \in R^{N \times d}$ to represent the position of an interaction in a sequence. The matrix for positional encoding is the same for interactions and exercise e_{t+1}. Matrices $\mathbf{M}$, $\mathbf{W}$ and $\mathbf{P}$ are learnt during training, while the inputs to further steps are the embedded matrices, with size $\hat{M} \in R^{N \times d}$ and $\hat{Q} \in R^{N \times d}$, where each row of these matrices is the sum of the interaction/exercise embedding and the positional one.

		exercise i		
		incorrect	**correct**	**total**
exercise j	**incorrect**	n_{00}	n_{01}	n_{0*}
	correct	n_{10}	n_{11}	n_{1*}
	total	n_{*0}	n_{*1}	n

Table 4.1: Contingency table. Values are taken from the original paper [18].

4.2.2. Self-attention layer

Self-attention is computed using as keys and values the interactions embeddings and as query the exercise embedding to predict. Each key-value pair produces a value embedding and all these embeddings are weighted summed by the feed forward layer, returning correctness prediction for each time-step. In Figure 4.2, we present the network architecture of SAKT.

4.3. Relation-Aware Self-Attention Knowledge Tracing

Relation-Aware Self-Attention Knowledge Tracing (RKT) [18] is a model extending SAKT to use additional information about the item (both textual and temporal). In Figure 4.3, we present the architecture of RKT.

It first generates for each word an embedding, using a parametric function $f : M \mapsto R^d$, where M is the dictionary of words and then computes smooth inverse frequency to generate exercise embeddings as follows:

$$E_i = \frac{1}{|s_i|} \sum_{w \in s_i} \frac{a}{a + p(w)} f(w) \tag{4.1}$$

where a is a trainable parameter, s_i is the text of i_{th} exercise and $p(w)$ is probability of word w. RKT computes cosine similarity between the exercise embeddings to generate a similarity matrix, whose coefficients represent similarity between the texts of two exercises.

RKT builds a contingency table as shown in table 4.1, by considering only the pairs of interacted exercises i and j, where j occurs before i in the learning sequence. Then it computes a second similarity matrix using Phi coefficient, a popular method to compute a measure of association between two binary variables, easy to interpret and explicitly penalizing when variables are not equal. $\phi_{i,j}$ coefficient between exercise i and exercise j

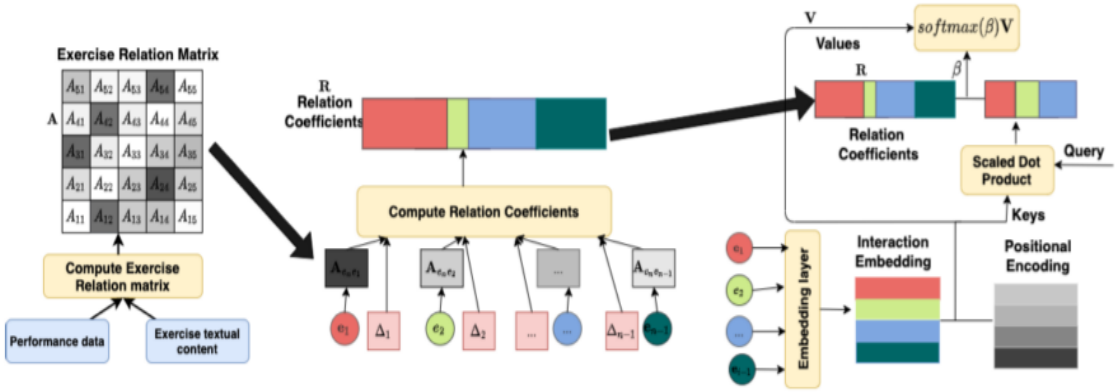

Figure 4.3: The overall architecture of RKT. After computing exercise relation matrix A, RKT generates relation coefficients between past exercises and next exercise, using A and time elapsed. Relation coefficients R are propagated to modify the attention weights. The image is taken from the original paper [18].

is computed as follows:

$$\phi_{i,j} = \frac{n_{11}n_{00} - n_{10}n_{10}}{\sqrt{n_{1*}n_{0*}n_{*1}n_{*0}}} \tag{4.2}$$

These two matrices are summed together to generate an unique exercise relation matrix A. In the end a third component models the tendency of each user to forget topics (called forgetting factor); considering exercise i we define $\delta_j = T_i - T_j$ and compute forgetting factor as:

$$R^T = [\exp\left(\frac{-\Delta_1}{S_u}\right), ..., \exp\left(\frac{-\Delta_{n-1}}{S_u}\right)] \tag{4.3}$$

This matrix is weighted summed with A to generate an unique relational matrix $\mathbf{R}$, representing collaborative, textual and temporal similarity between past exercise and the one to predict.

The model consists of a self attention layer with positional encoding (as SAKT), generating attention coefficients, weighted summed (with a trainable parameter) to $\mathbf{R}$. In the end a feed forward neural network computes outputs and a sigmoid function is used to predict probabilities.

4.4. Separated Self-AttentIve Neural knowledge Tracing

Separated Self-AttentIve Neural Knowledge Tracing (SAINT) [5] is a KT model using a novel Transformer-based architecture, enabling the encoder to embed relations between

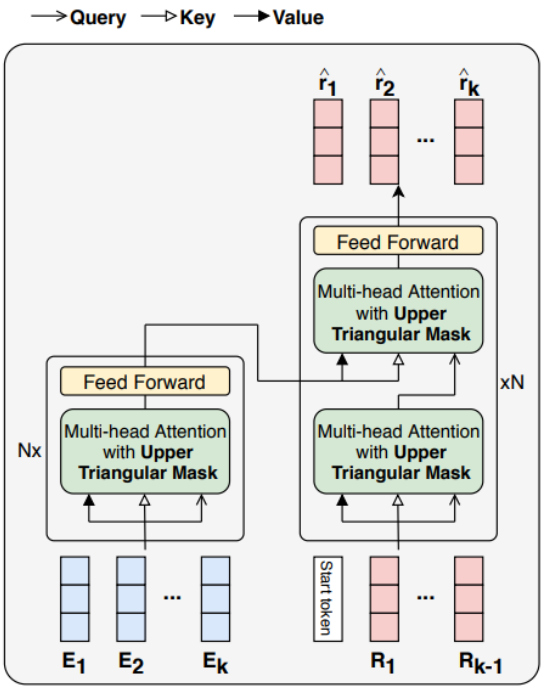

Figure 4.4: SAINT architecture. It separates the exercise sequence and the response sequence, applying to them the encoder and the decoder respectively. It can learn complex relations among exercises and responses. The image is taken from the original paper [5].

previous exercises and the one to predict, while decoder focus on embedding performance information (as the correctness of previous interactions). In Figure 4.4, we show SAINT architecture.

4.4.1. Embedding layer

Instead of embedding past exercises with the corresponding responses, as in SAKT, SAINT associates a latent vector to each skill id (or exercise category), item id and response value (binary in most cases). From $e_0, ..., e_{t+1}$, $s_0, ..., s_{t+1}$, and $r_0, ..., r_t$ it generates sequences of embeddings $e^e_0, ..., e^e_{t+1}$, $s^e_0, ..., s^e_{t+1}$, and $r^e_0, ..., r^e_t$.

Encoder inputs are the sum of the embeddings related to exercise, skill and a positional encoding: $E^e_i = e^e_i + s^e_i + p_i$, while decoder inputs are the sum of response embeddings with the same positional encoding of the encoder. Inputs to the decoder are shifted by one time-step in the future, to avoid attending to information about the future response.

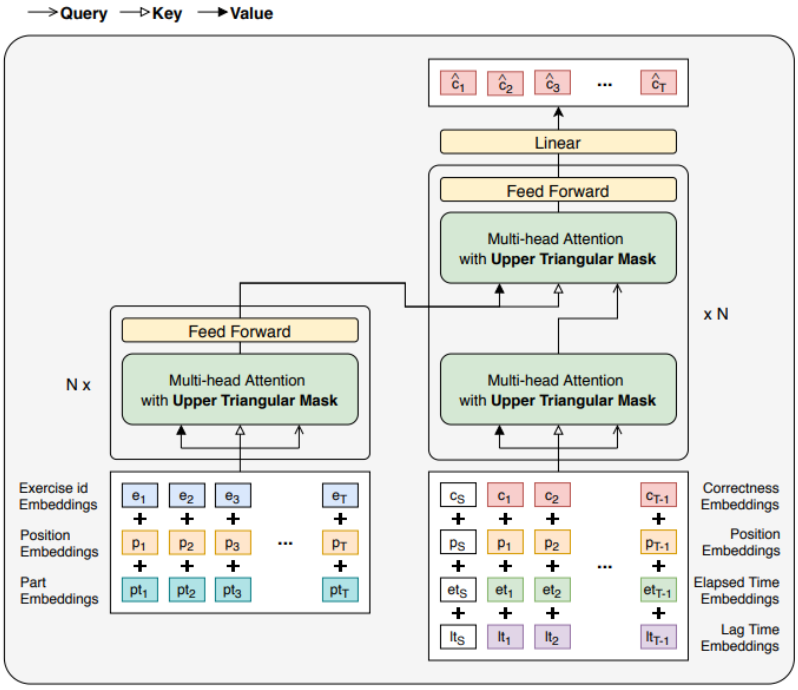

Figure 4.5: SAINT+ architecture, adding to the inputs of the decoder of SAINT+ an embedding for elapsed time and another for lag time. The image is taken from the original paper [25].

4.4.2. Transformer layer

Unlike the original Transformer architecture, SAINT masks inputs corresponding to information from the future for all multi-head attention layers to avoid invalid attending, ensuring that the prediction of $\tilde{r}_{t+1}$ depends only on exercises $e_0, ..., e_{t+1}$ and past responses $r^e_0, ..., r^e_t$ (not on r^e_{t+1}). In the end, a feed-forward layer, followed by a sigmoid operation is applied to the outputs of the decoder, producing a probability of correctness for each time step.

4.5. SAINT+

SAINT+ [25] is the successor of SAINT, adding temporal information to the inputs of the decoder. In particular, it uses: i) elapsed time, which is the time taken by a student to answer, and ii) lag time, the time interval between adjacent learning activities, to improve performances of the model, outperforming SAINT on the Ednet [4] dataset. These values are embedded in a categorical or continuous embedding and summed to SAINT decoder input (response embeddings and positional encoding).

Results from SAINT+ show clearly that the best way to insert temporal features is by summing them to response embeddings. Instead, it is not possible to affirm with certainty the best way to embed generic temporal features (categorically or continuously), since lag time seems to work better with categorical, while elapsed time works better with continuous embedding.

4.6. Exercise-Enhanced Recurrent Neural Network

Exercise-Enhanced Recurrent Neural Network (EERNN) [26] is a model composed of three parts: the first one produces exercise embeddings from exercise text, the second uses an RNN (or LSTM) to model the knowledge of users over time and the last one is responsible for the strategy to predict response.

4.6.1. Exercise embeddings

To the best of our knowledge, EERNN is the first model in KT using word2vec to generate word embeddings from texts. word2vec is used to associate to each word a word embedding; then each text is described as a sequence of word embeddings, given as input to a bidirectional LSTM, whose final hidden state represents the exercise embedding.

4.6.2. Student embeddings

EERNN embeds each interaction (e_t^i, r_t^i) of user i at time step t using the exercise embedding X_t generated from e_t^i text and concatenating it to a zero vector in a different way according to correctness of the response r_t^i:

$$\tilde{X}_t = \begin{cases} [X_t \oplus \mathbf{0}] & if\ r_t^i = 1 \\ [\mathbf{0} \oplus X_t] & if\ r_t^i = 0 \end{cases} \tag{4.4}$$

where $\oplus$ represents the concatenation operator between vectors. The sequence of embedded interactions $(\tilde{X}_1, \tilde{X}_2, ..., \tilde{X}_t)$ is now used as input to a RNN (or LSTM network) to model student knowledge over time.

Prediction layer

The prediction layer is responsible for using information from past interactions (h_1, ... , h_t) and next exercise embedding X_{t+1} to predict the correctness of the answer to the next exercise r_{t+1}.

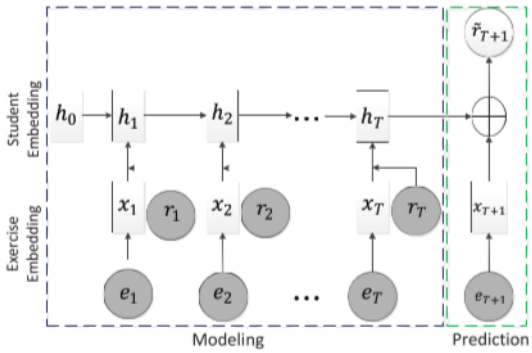

Figure 4.6: EERNN with Markov property architecture. The image is taken from the original paper [26].

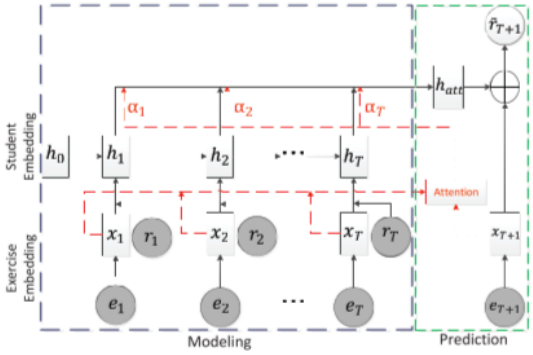

Figure 4.7: EERNN with attention mechanism architecture. The image is taken from the original paper [26].

For sequential prediction tasks, Markov Property is a well-known theory, which assumes that the next state depends only on the previous state and not on past ones. Following this assumption, EERNN with Markov property concatenates last hidden state h_t with target exercise embedding X_{t+1} and uses an FFNN with a single layer and sigmoid activation function to produce prediction from the concatenated vector.

As explained in Section 4.1.1, RNN and LSTM suffer the inability to represent whole past information in the last hidden state when the sequence is long. The alternative EERNN with Attention mechanism starts from the assumption that student state at time $t+1$ can be expressed as weighted combination of previous states, defining:

$$h_{att} = \sum_{j=1}^{t} \alpha_j h_j \tag{4.5}$$

$$\alpha_j = \cos(X_{t+1}, X_j) = \frac{\sum_{k=1}^{n} X_{t+1,k} X_{j,k}}{\sqrt{\sum_{k=1}^{n} X_{t+1,k}^2} \sqrt{\sum_{k=1}^{n} X_{j,k}^2}} \tag{4.6}$$

Final prediction is equal to EERNN with Markov property, replacing h_t with h_{att}.

5 | Proposed model architectures

In this chapter, we describe our work in detail explaining our considerations, the choices we make, and the proposed models we evaluate. In Section 5.1, we show six NLP techniques to produce embeddings from the text of the exercises, explaining the reasons behind them, the procedure to use them and their characteristics. In Section 5.2, we propose NLP-enhanced DKT, our first model, based on an LSTM network and able to receive as input the exercise embeddings. In Section 5.3, we present two models: Prediction Oriented Self-attentive knowledge Tracing with Multiplication (POST-M) and Prediction Oriented Self-attentive knowledge Tracing (POST). In particular POST is a Transformer based approach with an additional final decoder to refine the correctness prediction with the target exercise information (id and skill embeddings). POST is then modified in Section 5.4 to use as input the textual exercise embeddings explained in Section 5.1, similarly to what is done with NLP-enhanced DKT. Lastly, in Section 5.5, we present two ideas to create hybrid models, able to use at the same time multiple exercise embeddings coming from different NLP techniques (or, in general, from heterogeneous sources).

5.1. Generating exercise embeddings from texts

To efficiently make use of text to improve KT models, we consider the idea of decomposing the problem into two distinct parts: i) we create exercise embeddings from text and ii) we develop KT models able to receive them as inputs. In this section, we focus on the first part, explaining how to use six NLP techniques to produce exercise embeddings.

In Figure 5.1, we show a schema of the NLP methods we implement and evaluate to generate exercise embeddings. Starting from exercise text, we remove HTML tags, producing *plain text*, which is used as input for DistilBERT (§3.4.4), Sentence Transformer (§3.4.5) and BERTopic (§3.4.6) models. If we remove stopwords and lemmatize the words in *plain text* we create the so called *clean sentence*, which can be used as input for word2vec (§3.4.4), doc2vec (§3.4.5) and CountVectorizer (§3.4.2) methods.

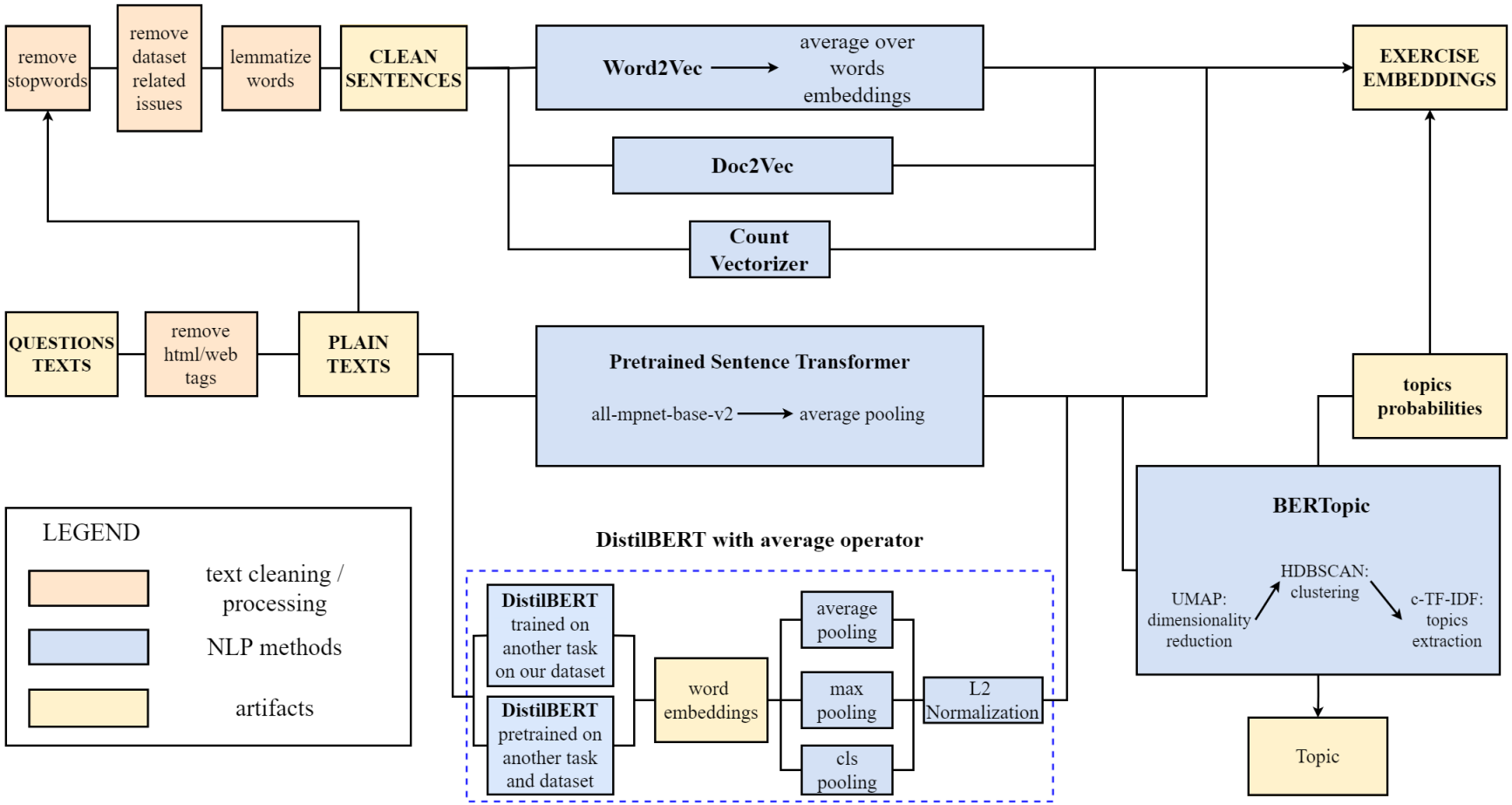

Figure 5.1: NLP methods to produce exercise embeddings. We have six possible paths from *questions texts* to produce an *exercise embeddings*, corresponding to the use of CountVectorizer, word2vec, doc2vec, Sentence Transformer, DistilBERT or BERTopic. Orange, blue and yellow blocks are related respectively to data processing, NLP methods and intermediate or final representations.

5.1.1. CountVectorizer

CountVectorizer (§ 3.4.2) is the simplest NLP method we apply to produce exercise embeddings, collecting all the n_w words, associating to each of them a unique index (from zero to $n_w - 1$) and representing a text as a vector with dimension n_w, where each value at position i is the number of times the word associated to index i appears in the text.

This method leads to vectors with a dimension equal to the number of words, usually too high to directly be used as inputs to KT models. Memory consumed by CountVectorizer with dense matrix representation is at least $n_d \cdot n_w$, where n_d is number of documents and n_w is number of different words. Since the dense matrix has few non-zero values for each row, we can reduce memory consumption by using a sparse implementation.

The dimensionality of a single exercise embeddings would still be high, so, to reduce it, words in exercise texts should be lemmatized (§ 3.4.1) and stopwords (§ 3.4.1) removed. Another possibility is to remove words appearing less than min_df times, more than max_df times, or maintain only the most frequent $max\_features$ words. In Section 6.7.1, we describe the variation of the number of words according to these three parameters.

5.1.2. Word2vec, DistilBERT and aggregation over words

Word2vec (§ 3.4.4) or DistilBERT (§ 3.4.4) are good choices to create fixed-length word embeddings since they have shown good performance on different NLP related tasks. We choose DistilBERT instead of BERT due to the smallest memory and time required, while the quality of the embeddings should not decrease significantly. However, they can not be used directly to improve KT, since describing a single text as a set of word embeddings would be extremely memory-consuming: memory used to describe n_d documents with an average n_w number of words would be $n_d \cdot n_w \cdot h_e$, where h_e is embedding vector dimension.

There are different solutions to use word embeddings to create document embeddings. Differently from EERNN, which uses an LSTM (§ 3.1.4) network to generate document embedding, we chose to apply a simple aggregation function over words dimension. We compute an exercise embedding (with the same fixed-length of word embeddings) as the average of the set of words embeddings generated from the words appearing in the exercise. Memory consumption becomes $n_w \cdot h_e$ to maintain word embeddings, plus $n_d \cdot h_e$ for exercise embeddings.

Computing average is a relatively fast operation while transforming texts into exercise embeddings can be time-consuming with DistilBERT. Another advantage of these two

methods is the possibility to have exercise embeddings with fixed-length, equal to word embeddings one. At the same time, averaging could lose some relevant features or make all exercise embeddings too similar. Since the average operator tends to produce similar embeddings as outputs, we recommend to apply normalization before using them for KT.

5.1.3. Doc2vec

Doc2vec (§ 3.4.5) is a good alternative to averaging word2vec embeddings, enabling to learn at the same document and word embeddings. This way we can use directly document embeddings as exercise embedding for KT. Doc2vec has the same advantages of using word2vec with average and, in addition, it can consider the order of words in the computation of exercise embedding. Memory consumption is identical to word2vec one since it keeps word and document embeddings.

5.1.4. Sentence Transformer

We denote as "Sentence Transformer" (§ 3.4.5) the Siamese network built on top of *all-mpnet-base-v2* Transformer, already implemented by SentenceTransformers[1] library, a python framework for state-of-the-art sentence, text and image embeddings. Sentence Transformer is a model creating exercise embeddings as outputs of the Pooling and Normalization layers applied to word embeddings generated with *all-mpnet-base-v2* transformer architecture. These exercise embeddings enable the reuse of knowledge about sentence similarity, so they should theoretically contain relevant information about the semantics of the exercise text.

5.1.5. BERTopic

Differently from previous NLP methods, BERTopic (§ 3.4.6) is not thought to produce document embeddings; instead, BERTopic can automatically discover topics of the texts and compute for each document a prediction of the topic or a probability vector, where each value at index i represents the probability of text to be about the topic associate to index i. We attempt to use the probability vector as an exercise embedding, checking if the predicted probabilities can be useful or not to improve KT. Our idea is to consider a "topic-oriented" approach, to see if it can provide useful insights and eventually replace the skill information, whenever these are not available.

[1] https://www.sbert.net/docs/pretrained_models.html

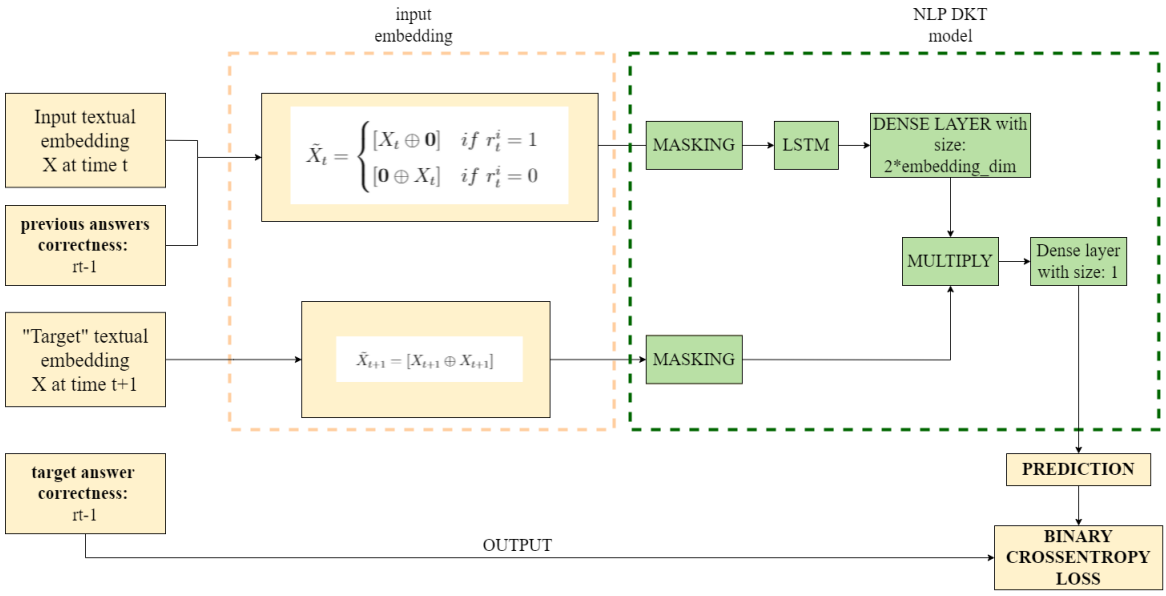

Figure 5.2: NLP-enhanced DKT model. At each time-step past exercise embedding (embedded with correctness) is given as input to the LSTM network, whose outputs are element-wise multiplied with the "target" exercise embedding. The result is passed to a dense layer producing "target" correctness prediction.

5.2. NLP-enhanced DKT

To check the utility of the different NLP methods, the first model we develop is NLP-enhanced DKT, which consists of an LSTM (§ 3.1.4) network, taking as inputs the exercise embeddings. In Figure 5.2, we provide a schema of NLP-DKT implementation.

The idea behind this model is to reproduce the DKT model with float vectors as input and outputs. Instead of one-hot encoding an integer variable, we embed interaction (e_t, r_t) at time step t as in EERNN (§ 4.6):

$$\tilde{X}_t = \begin{cases} [X_t \oplus \mathbf{0}] & if\ r_t^i = 1 \\ [\mathbf{0} \oplus X_t] & if\ r_t^i = 0 \end{cases} \tag{5.1}$$

where $\oplus$ represents the concatenation operator between vectors and X_t is exercise embedding associated to e_t, generated by a NLP method. The sequence of embedded interactions $(\tilde{X}_1, \tilde{X}_2, ..., \tilde{X}_t)$ is now used as input to the LSTM network.

Differently from EERNN, LSTM outputs $h_t \in R^{d_h}$ are passed to a Dense layer with sigmoid activation function, producing outputs $\tilde{X}_o \in R^{2 \cdot d_{embeddings}}$. First $d_{embeddings}$ elements of $\tilde{X}_o$ can be seen as positive knowledge representation, while the second half is negative

knowledge representation.

Now our model takes the "target" exercise embeddings X_{t+1} and compute element-wise multiplication to both the positive and negative half of $\tilde{X}_o$. In the end, a Dense layer is applied to compute a single value, representing answer $\tilde{r}_{t+1}$ correctness probability.

5.3. Prediction Oriented Self-attentive knowledge Tracing

In this section, we describe two new models: Prediction Oriented Self-attentive knowledge Tracing with Multiplication (POST-M) and Prediction Oriented Self-attentive knowledge Tracing (POST), both composed of a past exercise content encoder, a past performance decoder and a prediction oriented module.

First of all, embeddings are generated as in SAINT+: from $e_0, ..., e_{t+1}$; $s_0, ..., s_{t+1}$ and $r_0, ..., r_t$ three embedding matrices generates respectively sequences of embeddings $e^e_0, ..., e^e_{t+1}$, $s^e_0, ..., s^e_{t+1}$, and $r^e_0, ..., r^e_t$. The embedding matrices responsible for embedding exercise ids and skill ids are shared by exercise content encoder and prediction oriented module, while positional encoding is shared by exercise content encoder and performance decoder. It is important to explicitly say that all the following encoders and decoders use Masked Multi-Head Attention to avoid invalid attending, as in SAINT+.

5.3.1. Past exercise content encoder

The first component is an encoder similar to the encoder in SAINT+, where inputs are the sum of the embeddings related to exercise, skill and a positional encoding: $E^e_i = e^e_i + s^e_i + p_i$, but only for past time steps from $T = 0$ to $T = t$, padded with an initial start token. In Figure 5.3, we show the architecture of this component.

In SAINT+ the encoder has two objectives: learning which part of the embedding is useful and understanding relations between the past exercise embeddings and the one to predict. Instead, in our model the encoder can focus only on the first objective, understanding what is relevant and what is not.

5.3.2. Past performance decoder

Past performance decoder is equal to the decoder in SAINT+, embedding answer and time information from time step $T = 0$ to $T = t$, with an initial start token. Elapsed time is embedded using a continuous embedding, as suggested by SAINT+. This decoder uses

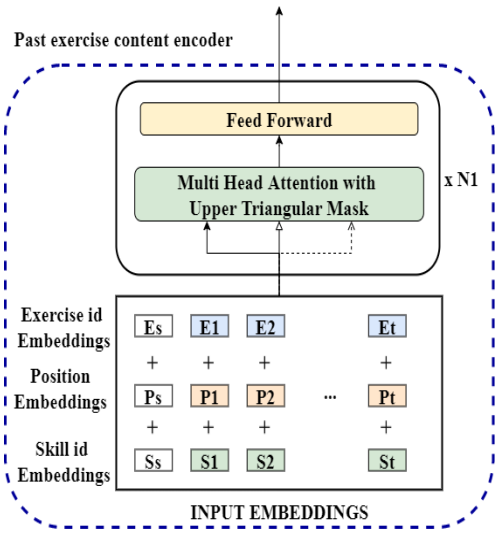

Figure 5.3: Past exercise content encoder module.

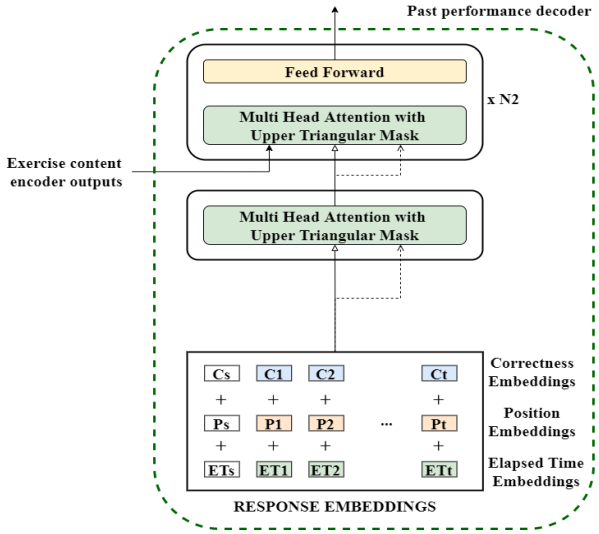

Figure 5.4: Past performance decoder module.

these embeddings as queries, while keys and values are the outputs from the past exercise content encoder. In Figure 5.4, we show the implementation of this component.

Differently from SAINT+, our models have the inputs for the encoder and the performance decoder aligned in time. We think this change can fasten the learning process and simplify learning dependencies. It is important to specify that this change does not enable invalid attending to future responses. For example, to predict correctness at time $T = 3$, the past exercise content encoder receives as input exercise content from $T = 0$ to $T = 2$ (plus initial start token) and the performance decoder receives as input responses for the same time interval. Only the prediction oriented module receives information about the content (not the response obviously) of exercise at time $T = 3$ to predict.

5.3.3. Prediction oriented module

The prediction oriented module is the main difference between our proposed models and SAINT+. No information about the time-step to predict $(t + 1)$ is given to the previous components, enabling them to focus only on embedding together past information about interacted exercise content and user response.

Instead, the third component is responsible of combining the performance decoder outputs y_{dec} and the target exercise embedding $E^e_{t+1} = e^e_{t+1} + s^e_{t+1}$ to predict the correctness r_{t+1}. POST-M and POST differ in the characteristics of the prediction oriented module.

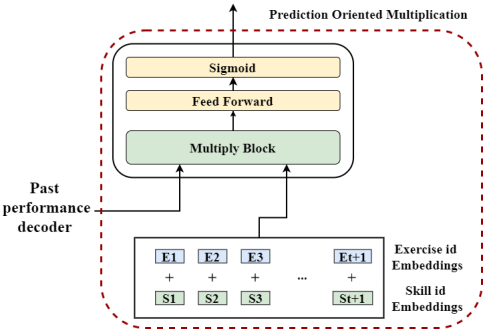

Figure 5.5: POST-M prediction module.

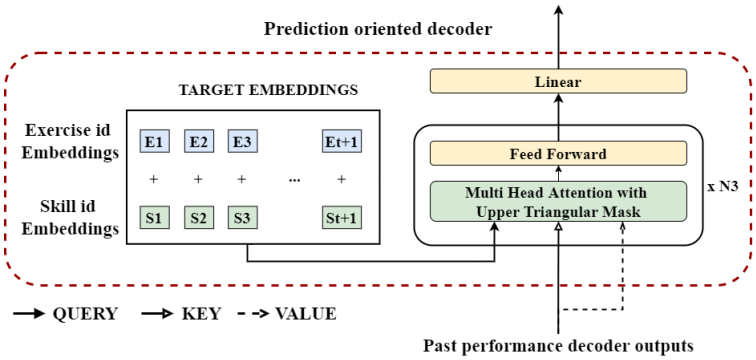

Figure 5.6: POST prediction module.

POST-M

POST-M computes Hadarmard product (also called element-wise product) of the inputs:

$$y_{t+1} = E^e_{t+1} \odot y_{dec} \tag{5.2}$$

where:

$$y_{t+1,k} = E^e_{t+1,k} \cdot y_{dec,k} \quad \forall k \tag{5.3}$$

In the end, outputs will pass through a feed-forward layer with sigmoid activation to compute correctness predictions. This method is simple and was born from the idea of replicating the approach used in NLP-enhanced DKT (§ 5.2). In Figure 5.5, we present the architecture of POST-M prediction module.

POST

Since the exercise content encoder and the performance decoder are based on self-attention layers, we consider if we can extend this approach to learn explicitly the relations between the past interactions and the target embedding. Therefore, POST is created by using as prediction oriented module another decoder, taking as query the target embedding E^e_{t+1} and, as key-value pairs, the outputs of past performance decoder, which summarize the knowledge evolution at each time-step from $T = 0$ to $T = t$. In the end, outputs will pass through a feed-forward layer with sigmoid activation function to compute correctness predictions.

In Figure 5.6, we present the architecture of POST prediction module; while in Figure 5.8 we show the complete architecture of POST model.

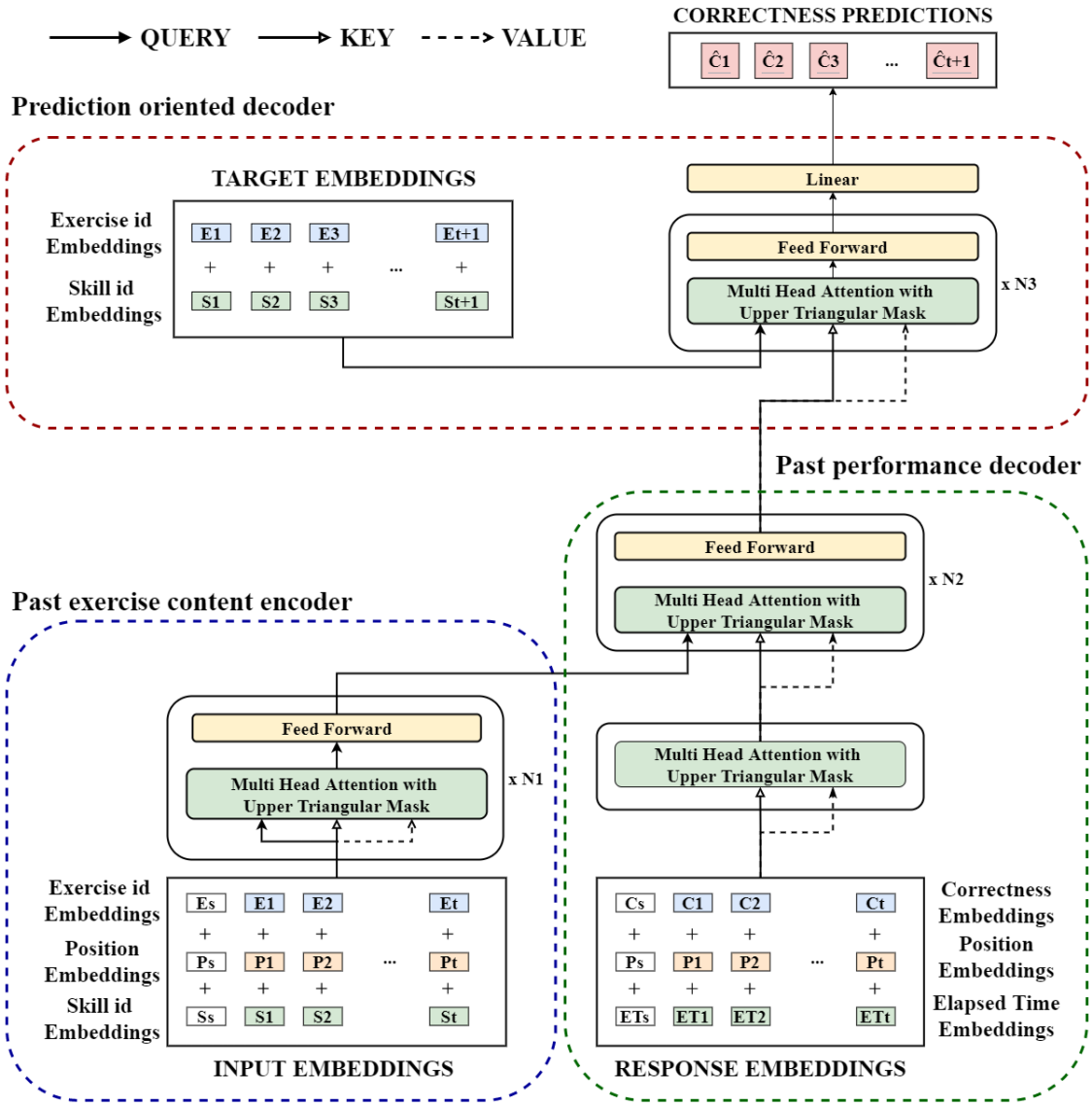

Figure 5.7: The representation of the architecture of POST. This model is composed of three modules: past exercise content encoder, past performance decoder and prediction oriented decoder. The first focuses on understanding what is relevant about past exercise content, while the second combine performance information (correctness and elapsed time) and encoder outputs. Lastly, the prediction oriented decoder combine exercise content embedding with the output of the performance decoder (representing information about past interactions) to make predictions.

5.4. NLP-enhanced Prediction Oriented Self-attentive knowledge Tracing

SAINT+ and POST can use as inputs only integer values, embedded later into float embeddings. Given a KT model, its extension, to use NLP exercise embeddings (which are float vectors) as inputs, is denoted as "NLP-enhanced". Similarly to how we create NLP-enhanced DKT from DKT, we develop NLP-enhanced POST: an extension of POST able to use as inputs the textual exercise embeddings explained in Section 5.1, which can be both integer (as CountVectorizer ones) or float vectors (word2vec, DistilBERT, doc2vec, Sentence Transformer and BERTopic ones). We identify two possible solutions to make use of textual embeddings:

1. Sum the textual embedding to the input embedding of POST.
2. Apply to each textual embedding a trainable Linear layer, computing $\tilde{X}$:

$$\tilde{X}(X) = f(W^T X + b) \tag{5.4}$$

   where $\tilde{X} \in R^{n_{dim}}$ and n_{dim} is the embedding size of the POST model. Then sum $\tilde{X}$ to the input embedding of POST.

The first approach is more straightforward but requires n_{dim} to be equal to the dimension of textual exercise embeddings, which is usually memory unfeasible. We use the second because, otherwise, CountVectorizer exercise embeddings could not be used due to memory limits.

In Figure 5.7, we show the total architecture of NLP-enhanced POST.

5.5. Hybrid approaches

Another contribution of our work is the possibility to use NLP-DKT and NLP-POST to create hybrid approaches, using multiple exercise embeddings from different sources at the same time. In this section, we propose some suggestions about how to use multiple NLP methods at the same time.

The hybrid approaches we present for NLP-DKT consist of parallelizing two or more of this model with different NLP methods and at a certain time combining the outputs. We can for example:

- compute the final prediction as weighted sum of the parallel models predictions

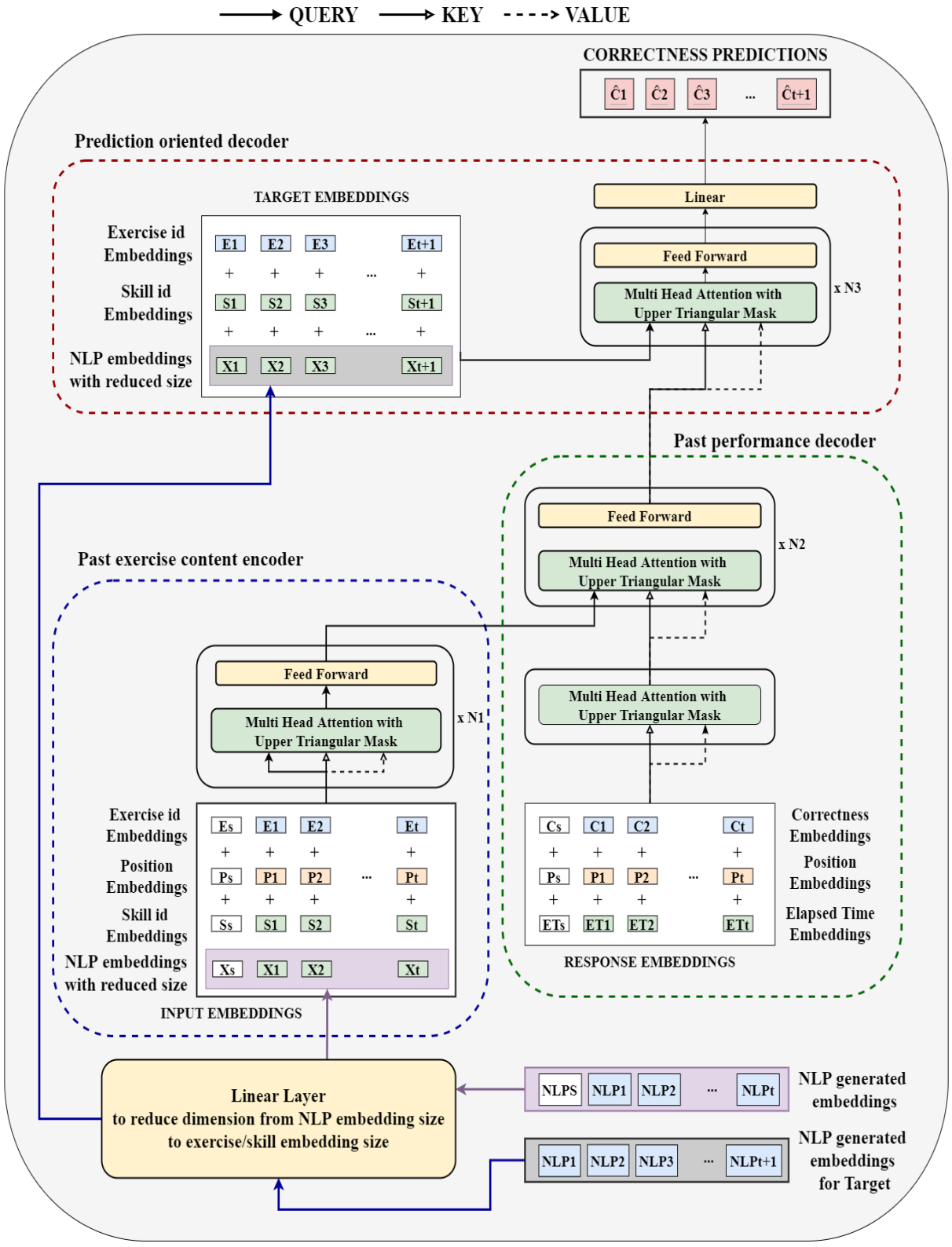

Figure 5.8: NLP-enhanced POST model extends POST adding a Linear layer responsible of reducing the size of NLP exercise embeddings to the dimension of the model. Then reduced NLP embeddings are summed to the inputs of past exercise encoder and prediction oriented decoder.

(shown in Figure 5.9);

- compute the final prediction applying a Dense layer with one output to the concatenation of the outputs of the Multiply blocks of the parallel models (shown in Figure 5.10).

Instead, the hybrid approach we propose for POST model is to modify inputs embeddings to past exercise content encoder and prediction decoder as the sum of NLP exercise embedding, skill id embedding, item id embedding and positional encoding:

$$E_{parallel} = E_{item\ id} + E_{skill\ id} + E_{positional} + \sum_{n=1}^{N} \tilde{X}_n(X_n) \tag{5.5}$$

$$\tilde{X}_n(X_n) = f(W_n^T X_n + b_n) \tag{5.6}$$

where X_n is the exercise embedding from n_{th} NLP method.

These hybrid approaches should not be considered as the best possible. They are examples of how to use at the same time skill, item and textual embeddings from different sources. We suggest to focus on hybrid approaches as a possible way to further improve our models, since there are a lot of possible techniques to create hybrids, other than parallelization.

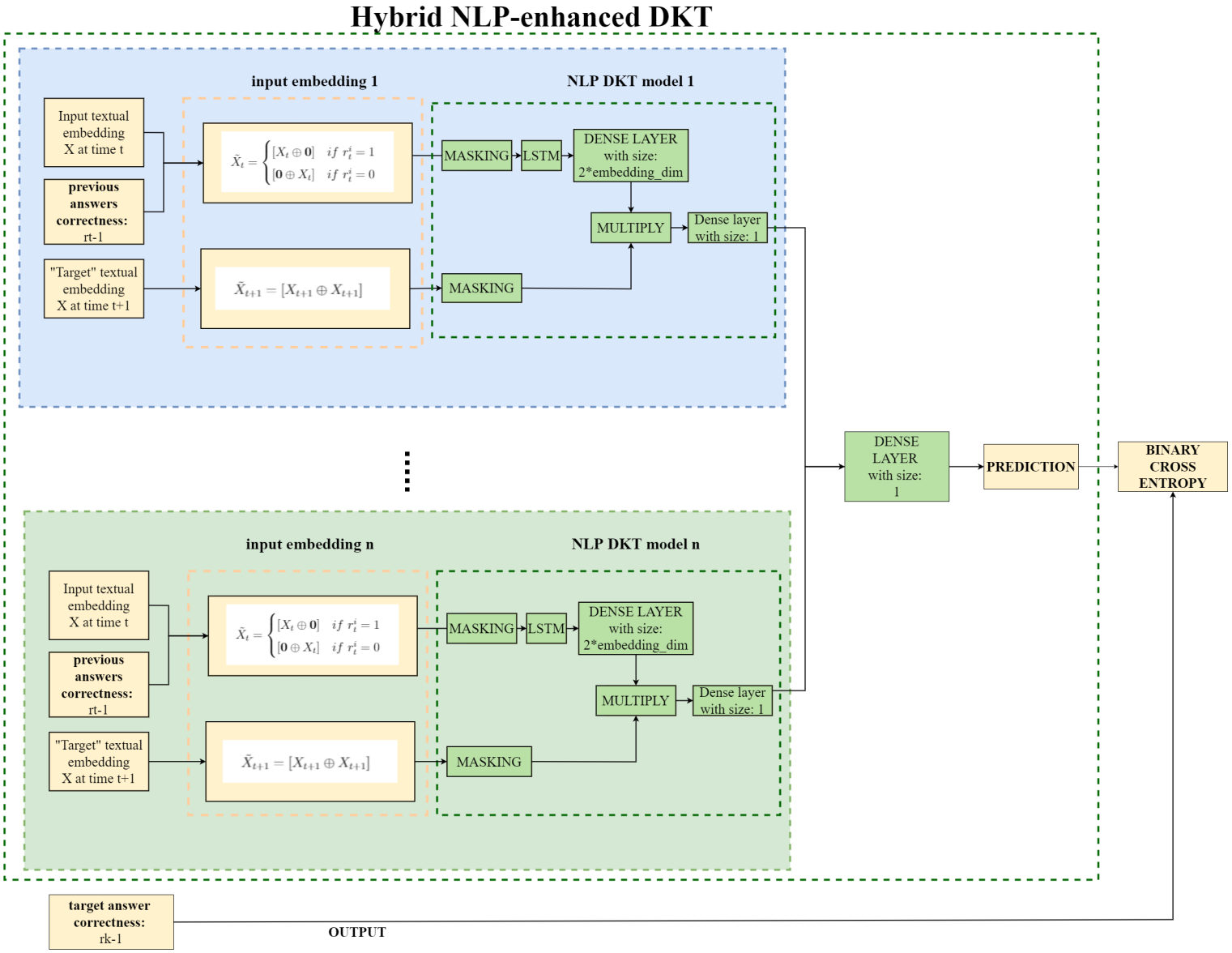

Figure 5.9: Hybrid approach to parallelize multiple NLP-enhanced DKT models, by computing final prediction as weighted sum of parallel predictions.

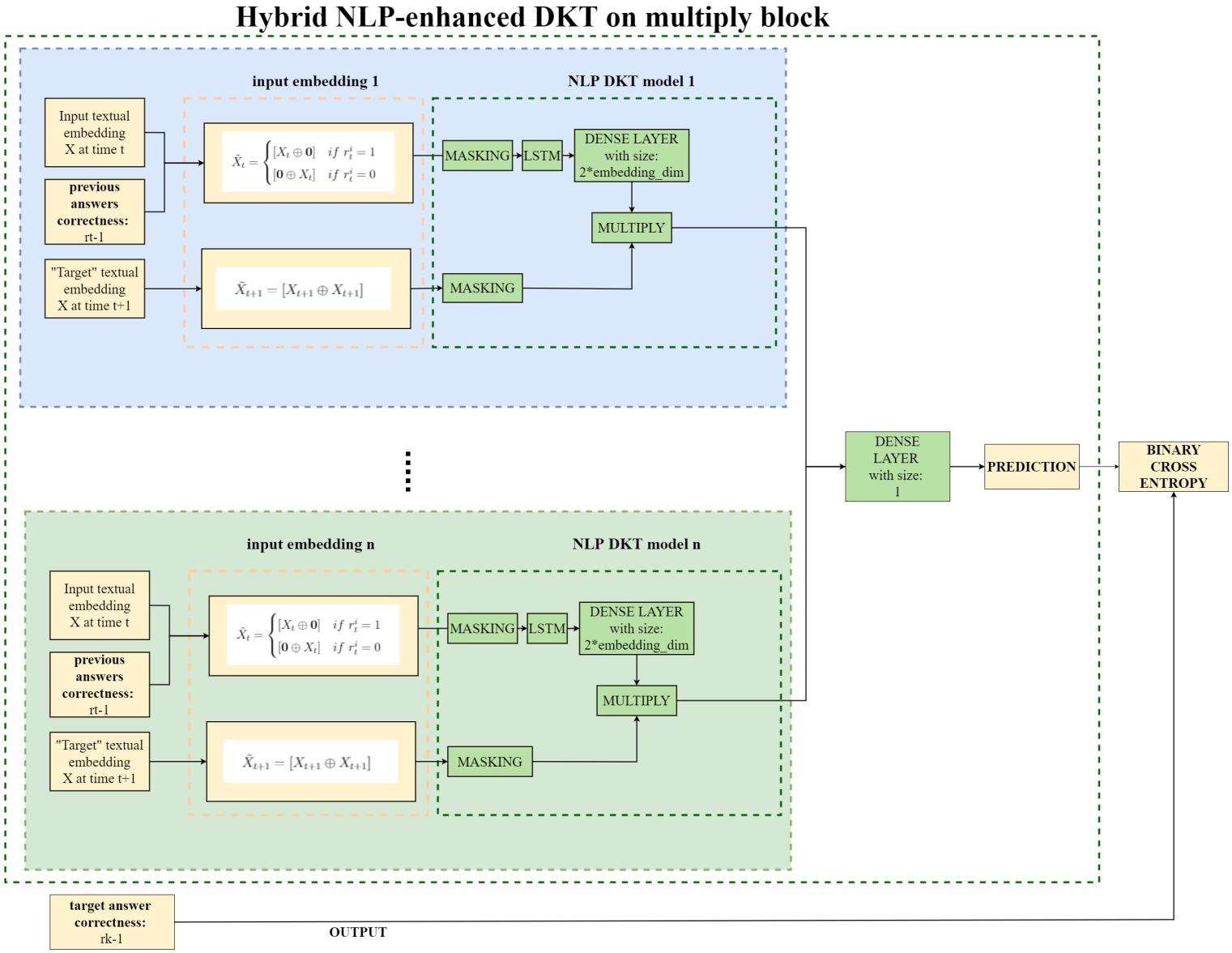

Figure 5.10: Hybrid approach to parallelize multiple NLP-enhanced DKT models, by applying a Dense layer with one output to the concatenation of the outputs of the Multiply blocks of the parallel models

6 | Experimental Setups

In this chapter, we present the setups we adopt for our experiments. In Section 6.1, we present the characteristics of the four datasets we choose; while in Section 6.2, we explain the processing operations to apply to the datasets. In Section 6.3, for each dataset we show the results of the data processing operations and describe the new common format. In Section 6.4, we explain the procedure to generate sequences of interactions from interactions records and the choices we make. Lastly, in Sections 6.5, 6.6 and 6.7, we present respectively how we split the data into training, evaluation and test sets, the loss and metrics we choose and the configuration of hyper-parameters we adopt.

6.1. Datasets

In this section, we describe the four datasets we use: ASSISTments 2009 (§ 6.1.1), ASSISTments 2012 (§ 6.1.1), Cloud Academy (§ 6.1.2) and Peking Online Judge dataset (§ 6.1.3). In KT, a dataset is mainly composed of interactions records, where an interaction is the act of submitting an answer r_t to an exercise question e_t at time t by the user U_t. An interaction record contains at least information about the user, the exercise id and the answer r_t and additionally can have some information about the skill s_t associated to the exercise or about the temporal features (such as start, end, elapsed or lag time).

Since we focus on textual information, we consider only datasets providing the text of exercises. We pre-process each dataset to produce a common representation of interaction and textual information, which is shared by all the datasets. Then, we can develop and evaluate our models to work on that common structure. To achieve that common representation, we consider only the interactions information shared by all the datasets for our KT models; the only exceptions are the elapsed time and skill id (when available) because they are widely adopted in the literature.

6.1.1. ASSISTments Datasets

ASSISTments[1] (AM) is a free online tutoring system, which helps teachers and students, providing resources and rapid feedback. It was developed in 2003 by Neil and Cristina Heffernan and offers teachers the possibility of retrieving useful content from open educational resources, while students can practice on assessments, receiving immediate feedback. Teachers can also receive reports on students' progress and performance.

We choose two datasets among ASSISTments ones for interactions: ASSISTments 2009[2] and ASSISTments 2012[3]; while texts dataset is in common for both the interaction datasets.

ASSISTments texts

We have access to the text of exercises, collected in a single file whose records are couples of:

- *problem_id*, denoting the unique identifier of the problem (exercise);
- *body*, which is the text of the exercise.

The number of problems is 179,969, some of them duplicated. Due to the presence of external references, we prefer not to remove duplicates to avoid assuming two exercises as being the same, while referenced entities could be different.

ASSISTments 2009 interactions

Interactions in ASSISTments 2009 (AM09) dataset are recorded as rows with a lot of fields, but we choose to consider only the ones available for all the datasets:

- *user_id*, which is the unique identifier for the student;
- *problem_id*, a unique identifier for the problem enabling to merge with texts dataset to retrieve the text of the exercise;
- *order_id*, chronological id enabling to order interactions;
- *skill_id*, the identifier of the skill associated with the problem. If multiple skills are associated with a problem, there would be multiple interactions, but since we do not need the skill, we merge by taking only the first one;

[1] https://new.assistments.org/

[2] `https://sites.google.com/site/assistmentsdata/home/2009-2010-assistment-data/skill-builder-data-2009-2010`

[3] `https://sites.google.com/site/assistmentsdata/datasets/2012-13-school-data-with-affect`

- *correct*, an integer value, representing if answer by the student was correct (1), or not (0). ASSISTments platform provides the possibility to ask for hint, in that case the answer is considered wrong.

ASSISTments 2009 dataset contains 346,860 interaction records about 4,217 different students.

ASSISTments 2012 interactions

Interactions for ASSISTments 2012 (AM12) dataset are recorded similarly to ASSISTments 2009 and we consider an interaction as composed of:

- *user_id*, which is the unique identifier for the student;
- *problem_id*, the unique identifier for the problem enabling to merge with texts dataset to retrieve the text of the exercise;
- *start_time*, timestamp describing when the problem starts;
- *end_time*, timestamp describing when the user submits the answer;
- *skill_id*, representing the identifier for the skill associated with the problem. If multiple skills are associated with a problem, there would be multiple interactions, but since we do not need the skill, we keep only the first one;
- *correct*, an integer value, identical to the one in ASSISTments 2009.

ASSISTments 2012 dataset contains 6,123,270 interaction records about 46,674 different students.

6.1.2. Cloud Academy Dataset

Cloud Academy[4] (CA) is an e-learning platform that helps companies to develop modern training paths for their employees, aiming to make them practical, impactful, and measurable. The dataset we use is part of their collection about IT technologies (such as AWS[5], Azure[6], GCP[7]) and similarly to ASSISTments is composed of two parts: interactions data and textual data.

[4]https://cloudacademy.com/
[5]https://aws.amazon.com/it/
[6]https://azure.microsoft.com/it-it/
[7]https://cloud.google.com/

Cloud Academy texts

The textual data provided by Cloud Academy consist of two files, one containing information about the text of the problem, the other about the text of the answers. We work only on the text of the problem, leaving how to use the text of answers an open problem.

The file containing text of the problems is structured as a collection of records with the following structure:

- *id*, which is the unique id of the problem;
- *description*, which is the textual description of the problem.

The number of problems is 15,523, each one with a unique text.

Cloud Academy interactions

Interactions provided by Cloud Academy are recorded as rows with multiple fields, but we choose to consider only:

- *_actor_id*, which is the unique identifier for the user;
- *question_id*, unique identifier for the problem enabling to merge with texts dataset to retrieve the text of the exercise;
- *_time_stamp*, the timestamp associated to each interaction;
- *elapsed_time*, representing the time elapsed between the instant the user can start reading the text and the instant the user submits the answer;
- *session_mode*, with three possible values: *exam*, *quiz*, or *study*. During *study* mode, the user can look at lessons before answering to a question, so we do not consider these interactions for KT;
- *correct*, 1 if the answer submitted by the student is correct, 0 otherwise.

Cloud Academy dataset contains 7,494,546 interaction records about 24,858 different students.

6.1.3. Peking Online Judge Dataset

Peking Online Judge[8] (POJ) is an online tool providing a dataset of coding problems and evaluating your submission. We use the datasets shared by the authors of RKT[9] [18].

[8] http://poj.org/
[9] https://drive.google.com/drive/folders/1LRljqWfODwTYRMPw6wEJ_mMt1KZ4xBDk

POJ data consists of two files: one with interactions logs and one with texts of exercises.

POJ texts

The file containing the text of the problems is structured as a collection of records with the following structure:

- *id*, which is an unique number associated to the text;
- *text*, which is the textual description of the problem.

POJ interactions

Interactions provided by POJ are recorded as rows with only four columns:

- *user*, which is the unique identifier for the user in a string or integer format;
- *problem*, the unique identifier for the problem enabling to merge with texts dataset to retrieve the text of the exercise;
- *submit time*, the timestamp associated to each interaction;
- *result*, which represents the result of the submission and can be *accepted*, *Compile error*, *run-time error*, *wrong answer*, *memory limit exceeded* or *time limit exceeded*. We generate our usual *correct* value by assuming 1 if *result* = *accepted*, 0 otherwise;

POJ dataset contains 996, 240 interaction records about 22, 916 different students.

6.2. Data processing

In this section, we describe the data processing operations in the same order we apply them to guarantee reproducibility.

6.2.1. Text cleaning

Since the main topic of our work is evaluating NLP methods, it is essential to consider only the datasets providing the text of exercises. Starting from the initial texts, we show in Figure 6.1 the procedure to generate *plain text* and *clean sentence*, which are used later as inputs for the NLP methods.

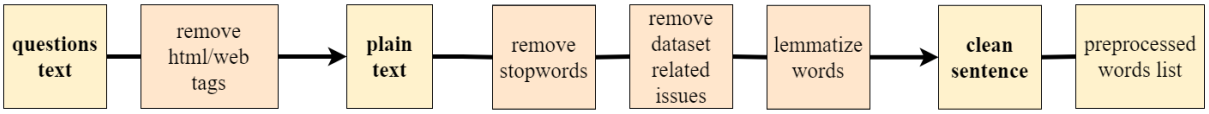

Figure 6.1: Text cleaning process.

Removing HTML tags, links, digits, punctuation and special characters

Since we are using datasets coming from online courses (or in general online environments), removing HTML tags is essential to guarantee the readability of the text string by the machine. However, sometimes exercise content is web-related and the name of tags can be useful to understand semantics; so, we decide not to delete the name of the removed tag, but to append it to the sentence. Then we need to use regular expressions to remove hashtags, links, special characters, punctuation and digits from the text. We refer to the output of this procedure as *plain text*. An example of the procedure is the following. From sentence:

```
<p> <span style="color: #0000ff;">Now you are ready to try the
original problem again.</span> If 900 students signed up to take
courses, how many will not be taking Biology, Algebra or Band?</p>
```

We obtain:

```
p nbsp span style color you are ready to try the original
problem again span if  students signed up to take
courses how many will not be taking biology algebra or band p
```

Removing stopwords, lemmatizing and tokenizing

As explained in Section 3.4.1, some NLP methods (such as CountVectorizer) require removing stopwords and lemmatizing the words to reduce the total number of words. Some datasets present a lot of misspelling errors, so we needed to use a spell checker to evaluate if each word exists. In a negative case, we need to decide whether to correct the word (using automatic tools), remove it or keep it.

Since these outputs are used by CountVectorizer and word2vec the sentences are then tokenized in a list of words. We refer to the output of this procedure as *clean sentence* or *list of words*. The previous sentence now becomes:

```
['p', 'span', 'style', 'color', 'ready', 'try', 'original', 'problem',
```

```
'span', 'students', 'signed', 'to', 'take', 'courses', 'many',
'taking', 'biology', 'algebra', 'band', 'p']
```

6.2.2. Removing interactions without text

After cleaning texts, we proceed to remove the interactions referring to an exercise without available text or whose text is too small to be useful (less than a certain min_w number of words). We choose $min_w = 2$ for any dataset. Differently from some previous works, we do not remove interactions related to a question without an associated skill. This information, essential to evaluate some models correctly, is not necessary for our work, since we extract content information directly from the texts.

6.2.3. Removing duplicated interactions

Differently from most previous works, we decide to consider the problem of duplicated interactions. Many datasets have users interacting multiple times with the same problem; it can be a direct consequence of the nature of the dataset (as for coding exercises, where a user compiles different times the code before finding a working solution) or the user is allowed to repeat the same exercise after some time.

We decide to keep only the first interaction of a user with a certain exercise, removing any subsequent one. This choice leads to different results compared to previous works since repeated interactions are usually easier to predict, since they respect some patterns (such as many compile errors in a row, or problem we already know answer, etc.).

6.3. Processed datasets

In this section we explain in detail how the data processing procedure (§ 6.2 is applied to each dataset. In particular, we describe how the number of interactions, problems, students and skills varies during the data processing procedure, in order to highlight particular characteristics of the datasets.

6.3.1. Processed ASSISTments 2009

To process ASSISTments 2009 dataset, we start processing text dataset as explained in Section 6.2.1, producing *plain text* and *clean sentence* for each exercise. Then we detect the duplicated interactions (keeping only the first one) and remove all the students with a single interaction; lastly, we compute the inner join between interactions and texts

datasets over the common key *problem_id*. After the inner join, ASSISTments 2009 has

Table 6.1: Evolution of the number of interactions, problems and users in ASSISTments 2009 dataset during data processing.

	Raw	Remove duplicates and users with a single interaction	Inner join with text dataset
# interactions	346,860	341,961	245,550
# problems	26,688	26,684	16,626
# students	4,217	4,097	3,836
# skills	150	150	96

16,626 problems with an average number of words in *clean sentence* equal to 47. It is important to note that in ASSISTments 2009 the *correct* mean $\bar{r}$ over the interactions is 0.6416, so the dataset is unbalanced. We consider as an additional baseline the majority prediction model, with an accuracy equal to the maximum between the correct mean $\bar{r}$ and $1.0 - \bar{r}$.

6.3.2. Processed ASSISTments 2012

To process ASSISTments 2012 dataset, we start processing the corresponding text dataset as explained in Section 6.2.1, producing *plain text* and *clean sentence* for each exercise. Then we detect the duplicated interactions (keeping only the first one) and remove all the students with a single interaction; lastly, we compute the inner join between interactions and text datasets over the common key *problem_id*.

Table 6.2: Evolution of the number of interactions, problems and users in ASSISTments 2012 dataset during data processing.

	Raw	Remove duplicates and single interaction	Inner join with text dataset
# interactions	6,123,270	5,894,203	5,892,571
# problems	179,999	179,969	179,919
# students	46,674	45,975	45,975
# skills	266	199	199

After the inner join between interactions and text datasets, ASSISTments 2012 has 179,919 problems with an average number of words in *clean sentence* equal to 18. In ASSISTments 2012 the *correct* mean $\bar{r}$ over the interactions is 0.6755, so the dataset is the most unbalanced. We consider as an additional baseline the majority prediction model, with an accuracy equal to the maximum between the correct mean $\bar{r}$ and $1.0 - \bar{r}$.

6.3.3. Processed Cloud Academy dataset

To process Cloud Academy dataset, we start from the text dataset as explained in Section 6.2.1, producing *plain text* and *clean sentence* for each exercise. Then, we remove from the dataset all the interactions with *session_mode* = *study*, the duplicated interactions, keeping only the first occurrence of each unique combination of *_actor_id* and *question_id*, and users with a single interaction. In the end we compute the inner join over the common keys: *question_id* for interaction dataset and *problem_id* for text one.

Table 6.3: Evolution of the number of interactions, problems and users in Cloud Academy dataset during data processing.

	Raw	Remove *study* mode	Remove duplicates and users with a single interaction	Inner join with text dataset
# interactions	7,494,546	3,723,691	2,381,050	2,381,030
# problems	13,605	13,268	13,267	13,262
# users	24,858	17,450	17,024	17,021

In table 6.3 we show how the number of problems and users does not significantly decrease, highlighting the reliability of the dataset. After the inner join between interactions and text datasets, Cloud Academy has $13,262$ problems, with an average number of words in *clean sentence* equal to 18. It is also important to highlight that this dataset is more balanced than ASSISTments ones; in fact, the mean value of *correct* is equal to 0.5455. We consider again as an additional baseline the majority prediction model.

6.3.4. Processed POJ dataset

To process POJ dataset, we start processing the corresponding text dataset as explained in Section 6.2.1, producing *plain text* and *clean sentence* for each exercise. First, we remove duplicated interactions, keeping only the first occurrence of each unique combination of *user* and *problem*, and users with a single interaction. In the end we compute the inner join over the common keys: *problem* for interaction dataset and *id* for text one.

In table 6.4, we can see how the number of interactions decreases drastically removing duplicates and merging with the text dataset. This is expected because i) the interactions represent the multiple attempts of a user to solve a coding exercise and ii) we have the text of only one-third of the exercises. It is important to note that datasets with users attempting many times to solve the same exercise are more difficult to deal with. In particular, if we have no additional information about the text of the answer or the time

Table 6.4: Evolution of the number of interactions, problems and users in POJ dataset during data processing.

	Raw	Remove duplicates and users with a single interaction	Inner join with text dataset
# interactions	996,240	126,663	49,638
# problems	2,750	2,746	774
# users	22,916	12,971	10,557

taken, it is extremely difficult to predict differently two consecutive answers of a user to the same exercise. Previous works using POJ dataset do not remove duplicated interactions. After the inner join between interactions and text datasets, POJ has 774 problems, with an average number of words in *clean sentence* equal to 143.

6.4. Generate sequences of interactions

After datasets are processed, they are saved with a common format, enabling the definition of a single method to load any of them. Each dataset is represented by two files: one for interactions and one for textual information. Interactions file consists of a list of recorded interactions, each one described by an unique couple of values (*user id*, *problem id)*, a *correct* value and some additional attributes (such as *timestamp*, *elapsed time* and *skill id*). If an attribute (such as *elapsed time*) is not available, the column is filled with 0 values. Textual information file associates each *problem id* with the *plain text* and the *clean sentence* generated during text processing.

Since the format is unique for all the processed datasets, the next operations are the same for each of them. After loading the processed files related to a dataset, we group the interactions according to the *user id* value, creating sequences of interactions related to the same *user id* and ordered according to *timestamp* value.

Then, for each sequence $(I_1, I_2, ...I_n)$, if $n < 5$ we discard the sequence, otherwise if $5 <= n <= N$ then the sequence is padded to length N, otherwise, if $n > N$, it is first divided into subsequences with length N and then the last one is padded (or discarded if $n < 5$). This enables the generation of sequences with the same length, a necessary condition for the self-attention based models. Whenever possible, we choose a maximum length $N = 500$, otherwise, if the KT model exceeds memory limits, we reduce N to 100. In table 6.5 we show for each dataset the number of padded sequences generated, the number of users, the average length of sequences before padding and how many sequences

are longer than N, both for $N = 100$ and $N = 500$.

All the KT models based on LSTM networks (such as DKT and NLP-enhanced DKT) use a maximum sequence length $N = 500$, while self-attention based models (such as SAINT+, POST-M, POST and NLP-enhanced POST) need to use $N = 100$, because of their larger memory consumption and memory limitations of our notebook.

Table 6.5: Number of users, chunks, padded sequences and average sequence length for each dataset.

N=100	AM09	AM12	CA	POJ
# Number of users	3,836	45,975	17,021	10,557
# Total number of chunks	1,409	41,935	16,534	59
# Sequences longer than N	657	14,722	6,251	30
# Average sequence length	64	128	139	5
# Number of padded sequences	4,625	83,034	32,535	2,160

N=500	AM09	AM12	CA	POJ
# Number of users	3,836	45,975	17,021	10,557
# Total number of chunks needed	66	3,239	1,016	1
# Sequences longer than N	66	2,383	721	1
# Average sequence length	64	128	139	5
# Number of padded sequences	3,346	4,5375	17,378	2,102

6.5. Data split and batches

Once the processed datasets are loaded and the padded sequences of interactions are created (as shown in Section 6.4), we need to split them into three datasets: training, validation and test. We decide to follow the usual practice of assigning 60% of the sequences to the training set, 20% to the validation set and the last 20% to the test set.

Sequences are grouped in batches, whose dimension changes according to the dataset and the model we are evaluating. For ASSISTments 2009 and POJ datasets, at the beginning of each epoch, we reorder the list of sequences and recreate the batches, while for ASSISTments 2012 and Cloud Academy this procedure would exceed memory limits.

The shuffle modifies only the order the sequences are given to the model for training, not the order of the interactions within a sequence.

6.6. Loss and metrics

The models are trained to minimize the Binary Cross Entropy loss (BCE), equal to:

$$BCE = -\frac{1}{N}\sum_{i=1}^{N} y_i * log(p(y_i)) + (1 - y_i) * log(1 - p(y_i)) \tag{6.1}$$

where, in KT, N is the number of interactions, y_i is the target correctness value to predict r_{t+1} and $p(y_i)$ is the prediction of the model $\tilde{r}_{t+1}$. We choose binary cross entropy, because it is the most used loss for binary classification problems, as it is the case of students' answer prediction.

For a binary classification problem, like students' answer prediction, the most common and useful metrics to evaluate the performance of a model are the Binary Accuracy and the Area Under the ROC Curve (AUC), equal to:

$$Binary\ Accuracy = \frac{TP + TN}{TP + TN + FP + FN} = \frac{TP + TN}{number\ of\ interactions} \tag{6.2}$$

where:

- True Positives (TP) is the number of correct answers correctly predicted;
- True Negatives (TN) is the number of wrong answers correctly predicted;
- False Positives (FP) is the number of correct answers mispredicted;
- False Negatives (FN) is the number of wrong answers mispredicted.

Binary Accuracy represents the ratio between the number of correct predictions and the total number of predictions.

The Receiver Operator Characteristic (ROC) curve (such as the examples in Figure 6.2) is a probability curve that plots the True Positive Rate (TPR) against the False Positive Rate (FPR) at various threshold values, where:

$$TPR = Recall = \frac{TP}{TP + FN} \tag{6.3}$$

$$FPR = \frac{FP}{FP + TN} \tag{6.4}$$

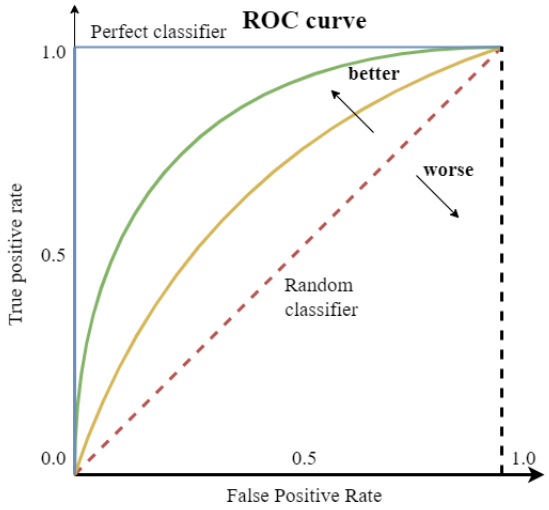

Figure 6.2: Four ROC curve examples; the blue one is the one of a perfect classifier, while the red one is from a random classifier.

The highest the Area Under the Curve (AUC), the better the model can distinguish between positive and negative classes. To better understand the performance of the models, we directly evaluate True Positive Rate (also called Recall) and Precision, equal to:

$$Precision = \frac{TP}{TP + FP} \tag{6.5}$$

In literature, KT models are evaluated mainly by comparing their AUC, so we primarily focus on this metric. Nevertheless, we decide to show other metrics too because they can possibly give more insights into the model abilities.

6.7. Hyper-parameters

Given the wide range of models and NLP techniques to evaluate, optimizing all the hyper-parameters would require too much time. Therefore, we decide to optimize only the learning rate of the training process. We compute the results using three different learning rates ($lr = 1e{-}3$, $lr = 1e{-}4$ and $lr = 1e{-}5$) for each model and report only the one with the highest AUC.

We fix the other hyper-parameters of the models to the following values:

- dropout rates equal to 0.3 for the LSTM networks (such as DKT and NLP-DKT) and to 0.2 for the self-attentive models (SAINT+, POST-M and POST).
- DKT is trained both to track questions or skills with a dimension *dim* equal to

200. If we track question ids, 200 is the largest number of units not exceeding memory limits; otherwise, if we track skill ids, the dimension of the input is smaller than 200, so there is no need to introduce more LSTM units. Instead, the self-attention based models are trained with an higher dimension equal to 768 (except with CountVectorizer, which exceeds memory with $dim > 256$).

- *Batch size* must be less or equal to 16 for ASSISTments 2012 or whenever we use CountVectorizer to generate exercise embeddings to avoid exceeding memory limits; Cloud Academy requires the batch size to be less or equal to 128 for the same reason. We decide to use 128 for Cloud Academy, ASSISTments 2009 and POJ.

Furthermore, the evaluated NLP methods use the following hyper-parameters:

- After processing, *clean sentence* includes mainly relevant words, so we try to reduce the least possible their number for CountVectorizer. The choice of CountVectorizer hyper-parameters is described in subsection 6.7.1.
- DistilBERT has pre-trained word embeddings and we do not specify any hyper-parameter to transform *plain text* into the document embedding. The word embedding size is 768.
- Overfitting the exercise embedding model is not a problem, so word2vec is trained for 200 epochs, using Skip-Gram architecture with word embedding size equal to 768 and window size equal to 5. The word embedding size is chosen equal to DistilBERT, while Skip-Gram should guarantee better results than CBOW.
- Doc2vec is trained for 100 epochs, using PV-DM architecture, with word embedding size equal to 768 and window size equal to 5 and 10 respectively for NLP-DKT and NLP-POST. Window size equal to 10 and PV-DM architecture are chosen because they provide small improvements to NLP-POST performance.
- Sentence Transformers uses the pretrained *all-mpnet-base-v2* Transformer model with size equal to 768.
- BERTopic uses *all-mpnet-base-v2* as Transformer to produce word embeddings. The only specified hyper-parameter is the maximum number of reduced topics, while the others are left to standard values decided by the author of the model. The maximum number of topics is fixed to 200, to keep it smaller than the other exercise embeddings sizes.

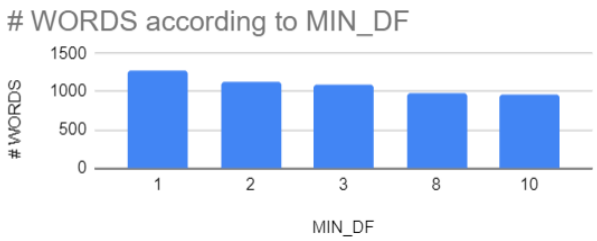

Figure 6.3: Number of words according to *min_df* for AM09 dataset.

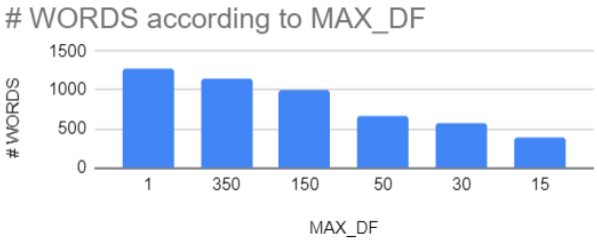

Figure 6.4: Number of words according to *max_df* for AM09 dataset.

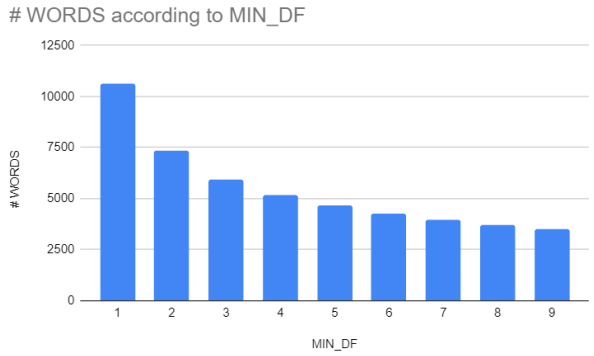

Figure 6.5: Number of words according to *min_df* for AM12 dataset.

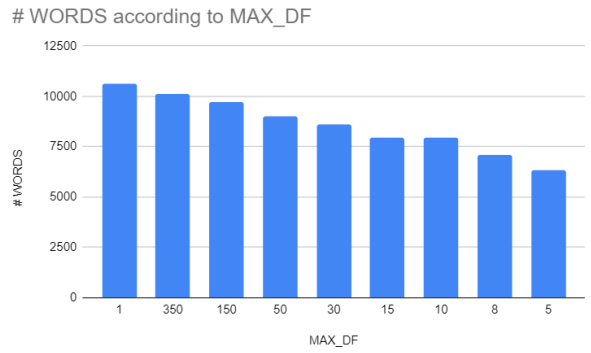

Figure 6.6: Number of words according to *max_df* for AM12 dataset.

6.7.1. CountVectorizer analysis

CountVectorizer is the only NLP method we evaluate that produces exercise embeddings with non-fixed length. The dimension of the exercise embeddings generated by CountVectorizer depends on the number of words, which varies according to the values of *min_df*, *max_df* and *max_features*. This variable length can result into exceeding time or memory limits, so we consider useful to analyze how the number of words varies according to *min_df* and *max_df* parameters. In our work, we make the following considerations:

- In Figures 6.3 and 6.4, we see how the number of words changes in ASSISTments 2009 processed texts according to *min_df* and *max_df* respectively. The initial number is only $1,282$, which is relatively small, so we can use directly all the words for CountVectorizer, without the need of modifying *min_df* and *max_df*.
- In Figures 6.5 and 6.6, we see how the number of words changes in ASSISTments 2012 processed texts according to *min_df* and *max_df* respectively. The initial number is $10,605$, which is unfeasible to work with according to memory limitations. So we decide to use a value $min\_df = 2$ to reduce the number of words for our

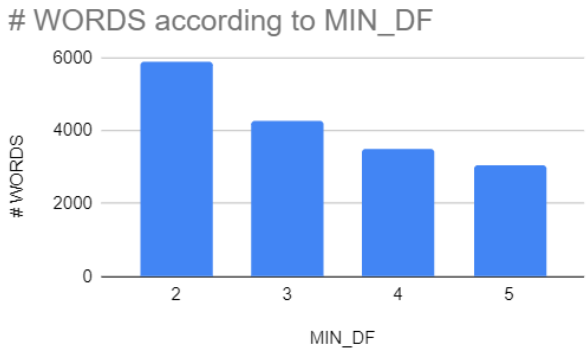

Figure 6.7: Number of words according to *min_df* for CA dataset.

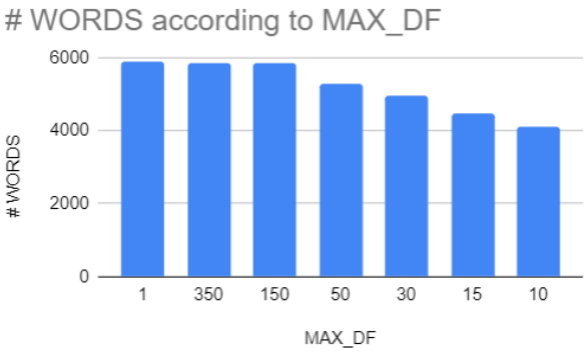

Figure 6.8: Number of words according to *max_df* for CA dataset.

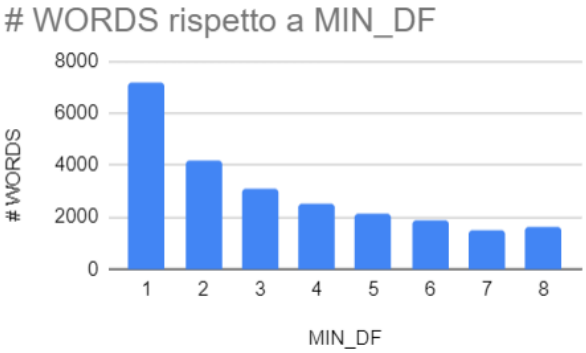

Figure 6.9: Number of words according to *min_df* for POJ dataset.

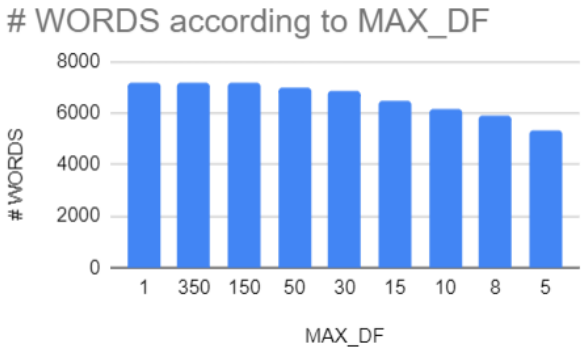

Figure 6.10: Number of words according to *max_df* for POJ dataset.

experiments and value $max\_features = 7,000$.

- We present in Figures 6.7 and 6.8 how the number of total words changes in CA processed texts according to min_df and max_df respectively. The initial number is $5,907$, so we decide to assign $min\_df = 2$ and $max\_features = 4,000$.
- Lastly, in Figures 6.9 and 6.10, we can see how the number of total words changes in POJ processed texts according to min_df and max_df respectively. The initial number in POJ processed texts is $7,210$, which is unfeasible to work according to our memory limitations. We fix $min\_df = 2$ again to reduce this number.

For most of the datasets we decide to fix $min\_df = 2$, because words appearing only in a single text are the least useful to understand relationships between exercises, because they do not connect exercises directly.

7 | Results

In this chapter, we report and discuss the results obtained from our experiments, dividing them into three sections. in Section 7.1, we show the results of POST-M and POST on the four datasets and compare them with the performance of baseline models DKT and SAINT+. in Section 7.2, we show and discuss the results of NLP-DKT and NLP-POST with the six NLP methods we already presented. Then, in Section 7.3, we present the performance of proposed hybrid approaches for NLP-DKT. Lastly, in Section 7.4, we highlight for each dataset the best performing models and we compute the total gain with respect to the best baseline.

7.1. POST-M and POST evaluation

In this section we evaluate four models (DKT, SAINT+, POST-M and POST) on four datasets (ASSISTments 2009, ASSISTments 2012, Cloud Academy and POJ). We add a fifth model, consisting of the majority prediction model, predicting always the mode of the correctness r_t. All the models are trained on the training set, using the validation set to apply early stopping. In the end, they are tested on the test set.

In this section, the columns appearing in the tables have the following meanings:

- *model* is the name of the evaluated model.
- *lr* is the learning rate used to train the model. We compute the results using three different learning rates ($lr = 1e^{-3}$, $lr = 1e^{-4}$ and $lr = 1e^{-5}$) for each model and report only the the one with the highest Area Under the Curve (AUC);
- *dim* represents the number of LSTM units in DKT model, while it is the embedding dimension (equal to the dimension of all the layers) in SAINT+, POST-M and POST;
- *drop* is the dropout rate of the hidden layers.
- N is the length of the padded sequences. DKT trained on skills requires less memory than the other models, so we can use $N = 500$;

model	hyper-parameters				metrics				
	lr	dim	drop	N	loss	ACC	AUC	Prec	Recall
majority	/	/	/	/	/	0.642	0.0	0.642	1.0
DKT skill	0.001	200	0.3	500	0.571	**0.711**	0.721	0.725	**0.896**
DKT question	0.01	200	0.3	100	0.739	0.669	0.693	0.732	0.77
SAINT+	1.00E-05	768	0.2	100	0.604	0.683	0.683	**0.882**	0.696
POST-M	0.001	768	0.2	100	0.596	0.682	0.711	0.829	0.713
POST	1.00E-05	768	0.2	100	0.575	0.707	**0.736**	0.862	0.725

Table 7.1: Results of the models on ASSISTments 2009.

- *loss* is the binary cross-entropy loss explained in Section 6.6;
- *ACC*, *AUC*, *Prec* and *Recall* are the binary accuracy, Area Under the ROC Curve, Precision and Recall respectively (§6.6).

We start considering the two datasets with the largest amount of data (ASSISTments 2012 and Cloud Academy) because they guarantee higher consistency between training, validation and test set. In tables 7.2 and 7.3 we report the results of the models on ASSISTments 2012 and Cloud Academy respectively. In these datasets, the DKT model shows the limitations of an LSTM network compared to attention-based models. The three self-attention based models show much higher AUC and ACC. SAINT+ performs slightly better than POST-M, probably due to the Multiply block, which reduces the power of the latter to learn complex relations between the target exercise embedding and the past ones. POST, replacing the multiply block with another decoder, removes this limitation and proves that a prediction oriented approach improves both ACC and AUC. We can see from Table 7.2 that the unbalanced nature of ASSISTments 2012 dataset leads to worse models predicting mostly the majority value (1), increasing Recall but reducing Precision. Instead, in table 7.3, we can see that improving the Recall metric leads a model to achieve better AUC and ACC values on Cloud Academy dataset than improving Precision.

From Tables 7.1 and 7.4, we can analyze the results of the models on ASSISTments 2009 and POJ respectively and understand the behaviours of the models with small amounts of training data. First of all, from ASSISTments 2009 results we can see that DKT

model	hyper-parameters				metrics				
	lr	dim	drop	N	loss	ACC	AUC	Prec	Recall
majority	/	/	/	/	/	0.676	0.0	0.676	1.0
DKT skills	0.001	200	0.3	500	0.557	0.717	0.725	0.737	**0.902**
SAINT+	0.0001	768	0.2	100	0.537	0.741	0.767	**0.884**	0.768
POST-M	1.00E-05	768	0.2	100	0.533	0.736	0.760	**0.884**	0.763
POST	0.0001	768	0.2	100	0.504	**0.754**	**0.790**	0.878	0.784

Table 7.2: Results of the models on ASSISTments 2012.

model	hyper-parameters				metrics				
	lr	dim	drop	N	loss	ACC	AUC	Prec	Recall
majority	/	/	/	/	/	0.546	0.0	0.546	1.0
DKT questions	0.01	200	0.3	100	0.47	0.634	0.694	**0.744**	0.636
SAINT+	0.0001	768	0.2	100	0.579	0.692	0.759	0.721	0.72
POST-M	0.0001	768	0.2	100	0.579	0.692	0.758	**0.744**	0.71
POST	0.0001	768	0.2	100	0.572	**0.699**	**0.769**	0.695	**0.736**

Table 7.3: Results of the models on Cloud Academy.

model	hyper-parameters				metrics				
	lr	dim	drop	N	loss	ACC	AUC	Prec	Recall
majority	/	/	/	/	/	0.552	0.0	0.552	0.0
DKT questions	0.0001	200	0.3	100	0.689	0.575	0.596	0.56	0.604
SAINT+	0.0001	768	0.2	100	0.65	0.618	0.661	0.557	0.61
POST-M	1.00E-05	768	0.2	100	0.642	0.627	0.68	0.546	**0.626**
POST	1.00E-05	768	0.2	100	0.647	**0.634**	**0.688**	**0.647**	0.622

Table 7.4: Results of the models on POJ.

(tracking skills) performs much better than SAINT+, highlighting the requirement for a larger amount of data to use SAINT+. We think this requirement derives mainly from the role of the encoder, responsible for learning at the same time how much attention to give to each part of the embeddings and the relations between the target exercise embeddings and previous ones. Instead, the structure of POST and POST-M, composed of three modules each one focusing on a single task learns faster and provides high AUC even with few training samples. It is important to highlight that DKT is the model with highest Binary Accuracy and Recall on this dataset. POJ dataset has instead no skill information available, so we can one-hot encode only question ids, resulting in low performance for DKT; while self-attention models provide better results. The results show that POST can improve significantly the four metrics while maintaining a loss similar to SAINT+. We consider this result as a demonstration of the better capabilities of POST to model answer correctness prediction task.

The results in tables 7.1, 7.2, 7.3 and 7.4 show that POST is the model providing the best AUC on each dataset.

7.2. NLP-enhancing KT models with textual exercise embeddings

In this section, we evaluate NLP-DKT and NLP-POST using as inputs the exercise embeddings produced by CountVectorizer (§5.1.1), word2vec (§5.1.2), DistilBERT (§5.1.2), doc2vec (§5.1.3), Sentence Transformer using *all-mpnet-base-v2* (§5.1.4) or BERTopic (§5.1.5). We compare NLP-DKT results with DKT baseline; while we compare NLP-POST only with POST, since the latter outperforms the AUC of SAINT+ and POST-M on all the datasets. Each table has rows ordered by ascending AUC metric, so the model in the last row is always the best performing for the dataset. We are not able to provide the results of NLP-DKT and NLP-POST using BERTopic embeddings for ASSISTments 2012 and Cloud Academy datasets, since the memory required to train the BERTopic model exceeds the limits of our notebook.

From the results, it is clear that NLP-enhancing is an effective idea to improve both DKT and POST since it improves their performance on all the datasets.

7.2.1. NLP-enhancing for ASSISTments 2009 dataset

In tables 7.5 and 7.6, we report the results of NLP-DKT and NLP-POST with six NLP methods on ASSISTments 2009 dataset. Except for BERTopic and DistilBERT respec-

model	parameters		metrics				
NLP method	**lr**	**dim**	**loss**	**ACC**	**AUC**	**Prec**	**Recall**
BERTopic	0.0005	768	0.604	0.685	0.682	0.697	**0.909**
Without NLP	0.001	200	0.557	0.717	0.726	0.738	0.903
Doc2vec	0.001	768	0.559	0.726	0.746	0.739	0.884
DistilBERT	0.001	768	0.553	0.723	0.753	0.735	0.887
Sentence Transformer	0.001	768	0.548	0.724	0.757	**0.754**	0.851
Word2vec	0.001	768	0.534	**0.738**	0.764	0.753	0.900
CountVectorizer	0.001	256	0.532	0.735	**0.778**	0.751	0.876

Table 7.5: Results of NLP-DKT and DKT on ASSISTments 2009.

model	parameters		metrics				
NLP method	**lr**	**dim**	**loss**	**ACC**	**AUC**	**Prec**	**Recall**
DistilBERT	1.00E-05	768	0.573	0.709	0.728	**0.863**	0.737
Without NLP	1.00E-05	768	0.5745	0.707	0.736	0.862	0.725
BERTopic	1.00E-05	768	0.571	0.712	0.742	0.855	0.735
Word2vec	0.0001	768	0.564	0.711	0.747	0.816	0.755
Doc2vec	0.0001	768	0.568	0.709	0.751	0.796	**0.761**
Sentence Transformer	0.0001	768	0.564	0.714	0.754	0.843	0.743
CountVectorizer	0.0001	256	0.5597	**0.715**	**0.756**	0.851	0.740

Table 7.6: Results of NLP-POST and POST on ASSISTments 2009.

tively for NLP-DKT and NLP-POST, all the other methods provide an improvement to the baseline. In particular, this is the only dataset where a NLP-DKT model outperforms the AUC of all the NLP-POST models. We believe that the reason is again the small number of samples available, which favours LSTM models and penalize the performance of attention based ones.

The best model for the dataset is NLP-DKT with CountVectorizer, followed by NLP-DKT with word2vec. It makes sense that the simplest NLP method (CountVectorizer) provides the best results with NLP-DKT, because its embeddings just track the presence of words in texts, without aiming to understand the semantics of the text. Learning the relationships between semantic embeddings, as DistilBERT, is more complex and requires a larger dataset or a more complex model. Despite performing the best with CountVectorizer, NLP-POST provides similar results with most of the NLP methods. In the end, it is important to note that BERTopic is particularly unuseful for small datasets because learning relevant skills with a small number of texts is more difficult.

7.2.2. NLP-enhancing for ASSISTments 2012 dataset

In tables 7.7 and 7.8, we report the results of NLP-DKT and NLP-POST on ASSISTments 2012 dataset. Before making considerations about the results, it is important to note that ASSISTments 2012 has an average sequence length equal to 128 and a large number of sequences for training, validation and testing. We analyze the results of NLP-DKT and NLP-POST separately:

- NLP-DKT improves DKT using any NLP method except CountVectorizer confirming the ability of NLP methods to embed ASSISTments texts. In particular, Sentence Transformer is the best performing model, while Count Vectorizer is the worst one. Differently from ASSISTments 2009, having more interactions and longer sequences enables the models to understand the embeddings generated by transformer-based NLP approaches (such as Sentence Transformer and DistilBERT).
- NLP-POST can provide only a very small improvement to the AUC of POST using CountVectorizer. We discuss in Section 7.2.5 the possible reasons.

model	parameters		metrics				
NLP method	**lr**	**dim**	**loss**	**ACC**	**AUC**	**Prec**	**Recall**
CountVectorizer	0.001	64	0.565	0.717	0.689	0.721	**0.948**
Without NLP	0.001	200	0.557	0.717	0.726	0.738	0.903
Doc2vec	0.001	768	0.5356	0.736	0.748	0.750	0.919
DistilBERT	0.001	768	0.530	0.736	0.760	0.751	0.912
Word2vec	0.001	768	0.526	0.739	0.760	0.753	0.916
Sentence Transformer	0.0001	256	0.510	**0.749**	**0.780**	**0.775**	0.890

Table 7.7: Results of NLP-DKT and DKT on ASSISTments 2012.

model	parameters		metrics				
NLP method	**lr**	**dim**	**loss**	**ACC**	**AUC**	**Prec**	**Recall**
Doc2vec	0.0001	768	0.511	0.751	0.784	**0.920**	0.762
Sentence Transformer	1e-5	768	0.510	0.751	0.784	0.903	0.770
DistilBERT	1e-5	768	0.514	0.750	0.785	0.897	0.770
Word2vec	0.0001	768	0.511	0.748	0.786	0.850	**0.793**
Without NLP	0.0001	768	0.504	0.754	0.790	0.878	0.784
CountVectorizer	0.0001	256	0.500	**0.757**	**0.793**	0.896	0.779

Table 7.8: Results of NLP-POST and POST on ASSISTments 2012.

7.2.3. NLP-enhancing for Cloud Academy dataset

In tables 7.9 and 7.10, we report the results of NLP-DKT and NLP-POST on Cloud Academy dataset. NLP-enhancing produces nearly no improvements to the performance of DKT and POST on Cloud Academy dataset. In fact, NLP-DKT shows worse performance than DKT with any exercise embeddings, while NLP-POST model produces results near to POST with any exercise embedding. We believe that these results are due to the textual content of the dataset, which is subject-specific (i.e. about IT technology). Understanding the similarities and the differences between exercise texts with these characteristics becomes more challenging and requires specific focus and effort. Being unable to easily understand the relationships, NLP-POST focuses only on the item id

model	parameters		metrics				
NLP method	**lr**	**dim**	**loss**	**ACC**	**AUC**	**Prec**	**Recall**
CountVectorizer	0.0001	256	0.647	0.575	0.629	0.684	0.617
Word2vec	0.001	768	0.5903	0.578	0.640	0.701	0.573
Doc2vec	0.001	768	0.5577	0.607	0.656	0.708	**0.654**
Sentence Transformer	0.001	768	0.5928	0.603	0.664	0.706	0.639
DistilBERT	0.0001	768	0.5506	0.596	0.665	0.719	0.596
Without NLP	0.001	200	0.4704	**0.634**	**0.694**	**0.744**	0.636

Table 7.9: Results of NLP-DKT and DKT on Cloud Academy dataset.

model	parameters		metrics				
NLP method	**lr**	**dim**	**loss**	**ACC**	**AUC**	**Prec**	**Recall**
Sentence Transformer	1e-5	768	0.572	0.698	0.766	0.731	0.722
DistilBERT	0.0001	768	0.571	0.699	0.768	0.746	0.716
Word2vec	0.0001	768	0.573	0.699	0.768	**0.761**	0.708
Doc2vec	0.0001	768	0.573	0.699	0.768	0.690	**0.737**
Without NLP	0.0001	200	0.572	0.699	0.769	0.695	0.736
CountVectorizer	0.0001	128	0.568	**0.703**	**0.771**	0.741	0.725

Table 7.10: Results of NLP-POST and POST on Cloud Academy.

model	parameters		metrics				
NLP method	**lr**	**dim**	**loss**	**ACC**	**AUC**	**Prec**	**Recall**
Without NLP	0.001	200	0.689	0.575	0.596	0.560	0.604
BERTopic	0.001	32	0.671	0.594	0.630	0.588	0.563
Word2vec	0.001	768	0.654	0.610	0.657	0.619	0.549
CountVectorizer	0.0001	256	0.661	0.608	0.658	0.608	0.521
Doc2vec	0.001	768	0.662	0.608	0.659	0.603	**0.657**
DistilBERT	0.0001	768	0.648	0.621	0.665	0.609	0.582
Sentence Transformer	0.001	768	0.649	**0.628**	**0.670**	**0.634**	0.597

Table 7.11: Results of NLP-DKT and DKT on POJ dataset.

embeddings.

7.2.4. NLP-enhancing for POJ dataset

In tables 7.11 and 7.12, we report the results of NLP-DKT and NLP-POST on POJ dataset. NLP-DKT shows improvements with respect to DKT with every exercise embeddings. In particular, Sentence Transformer provides NLP-DKT the highest Acc, AUC and Precision. NLP-POST improves the already high ACC, AUC, Precision and Recall of POST; differently from NLP-DKT, Sentence Transformers performs worse than the other methods. NLP-POST using word2vec or CountVectorizer are the best performing models for the POJ dataset.

7.2.5. Considerations about NLP-enhancing

The results confirm the utility of using texts to improve KT. In particular, NLP-enhancing shows the best utility to improve performance with small datasets. The small amount of available data about interactions usually limits the ability of the model to learn relations between exercises; instead, knowing the textual exercise embedding guarantees additional information, so the model can learn relations faster, needing fewer interactions. The results of NLP-enhanced models on ASSISTments 2009 and POJ datasets confirm this consideration.

Instead, for ASSISTments 2012 and Cloud Academy all the NLP-POST models perform

model	parameters		metrics				
NLP method	**lr**	**dim**	**loss**	**ACC**	**AUC**	**Prec**	**Recall**
Sentence Transformer	0.0001	768	0.650	0.622	0.672	0.5299	0.622
Without NLP	0.0001	768	0.647	0.634	0.688	0.647	0.623
Doc2vec	0.0001	768	0.637	0.639	0.698	0.712	0.620
BERTopic	0.0001	768	0.631	**0.648**	0.701	0.558	0.649
DistilBERT	1.00E-05	768	0.631	**0.648**	0.702	0.639	**0.672**
CountVectorizer	0.0001	128	0.649	0.635	**0.703**	**0.749**	0.585
Word2vec	0.0001	768	0.6281	0.644	**0.703**	0.551	0.651

Table 7.12: Results of NLP-POST and POST on POJ dataset.

the same, meaning that the textual exercise embeddings are simply ignored by the model. We think that the reasons behind this problem are the following ones:

1. POST model is already able to learn relations among exercises from a large number of sequences of interactions available. So the possible margin of improvement of the performance for large datasets is small and difficult to achieve. In general, adding information about the text to an already strong and complex model is difficult.

2. We argue that the linear layer to reduce the dimension of the embeddings can be the limitation, in particular for float exercise embeddings (such as Sentence Transformer, word2vec, etc.). It could be interesting to replace the linear layer with another self-attention based module responsible for learning attention weights to create reduced textual embeddings.

3. Another possibility explaining the limited results is the fact that we sum NLP reduced exercise embeddings to skill and item embeddings. Instead, it could be interesting to duplicate the past exercise content encoder: one focusing on skill and item embeddings, while the other can focus only on NLP reduced exercise embeddings. This duplication can possibly solve the lack of "attention" that NLP-POST models give to textual embeddings.

4. Another necessary consideration is that the number of average words per text is much lower in ASSISTments 2012 and Cloud Academy (equal to 18) than in ASSISTments 2009 and POJ (respectively equal to 47 and 143). The small number of

words per text can be another reason for the low results of NLP-enhancing on these datasets.

7.2.6. Comparison of NLP methods

To understand the capabilities of the NLP methods, we analyze both the AUC and the binary cross-entropy loss of the evaluated models. For NLP-DKT the best model on ASSISTments 2009 is CountVectorizer (or eventually word2vec), while on each other dataset is Sentence Transformer or DistilBERT. In our opinion, CountVectorizer is a better model when texts contain words with precise meaning (such as *equation*, *plus*, *exponential* in ASSISTments), while Sentence Transformer, word2vec and DistilBERT can be better with long texts (as in POJ) or with large datasets, because they are more powerful and can understand the semantic of the texts. We do not understand the reason, bu it is particularly interesting that doc2vec or word2vec provide the highest Recall with most of the datasets.

For NLP-POST we consider only the results on ASSISTments 2009 and POJ because NLP-POST suffers the problems presented in Section 7.2.5. We can affirm that with these small datasets all the NLP-POST models have similar performance.

In conclusion, from the results of our experiments on NLP-enhancing, we can only conclude that BERTopic is not suitable for answer correctness prediction, while the performance of the other methods varies according to the dataset characteristics. No method can be considered as the best one.

7.3. Hybrid approaches evaluation

In this section, we present the results on the four datasets of the hybrid approaches presented in Section 5.5. For each dataset, we consider from two to four NLP-DKT models and parallelize them to create the hybrid models using the two approaches explained in Section 5.5. The model with the best AUC between NLP-DKT models (or DKT) is the new baseline we compare our models with. In tables 7.13 and 7.15, producing the final prediction as a weighted combination of the parallel predictions performs better than the other approach. In table 7.16, the two approaches produce the same AUC, but summing parallel predictions provides a smaller loss and higher ACC.

The main consideration we can derive from the results is the utility of creating hybrid approaches, improving AUC on all the datasets.

model	params	metrics				
	lr	loss	ACC	AUC	Prec	Recall
weighted sum of Multiply vectors	0.001	0.532	0.733	0.777	0.755	0.861
weighted sum of predictions	0.0001	0.528	**0.742**	**0.784**	**0.763**	**0.864**

Table 7.13: Results of the two hybrid approaches to parallelize multiple NLP-DKT models (CountVectorizer, DistilBERT and word2vec) on ASSISTments 2009 dataset.

model	params	metrics				
	lr	loss	ACC	AUC	Prec	Recall
weighted sum of Multiply vectors	0.0001	0.516	**0.745**	**0.775**	0.762	**0.906**
weighted sum of predictions	0.0001	0.520	0.743	0.772	**0.766**	0.892

Table 7.14: Results of the two hybrid approaches to parallelize multiple NLP-DKT models (DistilBERT, Sentence Transformer and word2vec) on ASSISTments 2012 dataset.

model	params	metrics				
	lr	loss	ACC	AUC	Prec	Recall
weighted sum of Multiply vectors	0.0001	0.525	0.616	0.686	**0.737**	0.613
weighted sum of predictions	1e-5	0.638	**0.666**	**0.699**	0.669	**0.754**

Table 7.15: Results of the two hybrid approaches to parallelize multiple NLP-DKT models (CountVectorizer, DistilBERT and doc2vec) on Cloud Academy dataset.

model	params	metrics				
	lr	loss	ACC	AUC	Prec	Recall
weighted sum of multiply vectors	0.0001	0.652	**0.653**	**0.696**	**0.681**	0.631
weighted sum of predictions	1e-5	0.647	0.639	**0.696**	0.660	**0.632**

Table 7.16: Results of the two hybrid approaches to parallelize multiple NLP-DKT models (DistilBERT, BERTopic and word2vec) on POJ dataset.

7.4. Best performing models

In this section, we provide a summarized view for each dataset of the results of the different models. In particular, in tables 7.17, 7.18, 7.19, 7.20, we present the results of the best performing:

- baseline (DKT or SAINT+);
- model using questions ids or skill ids;
- NLP-enhanced model and respective NLP method;
- hybrid approach for NLP-DKT.

We can now compute the total gain of AUC our proposed best model provides to the best baseline (DKT or SAINT+) for each dataset:

- For ASSISTments 2009 dataset, the best model consists of a hybrid model composed of three parallel NLP-DKT models (respectively using CountVectorizer, Sentence Transformer and word2vec embeddings) and producing the final prediction as a weighted combination of the parallel predictions. This model improves AUC from 0.721 (DKT) to 0.784, with an improvement equal to 0.063, corresponding to a percentage improvement of 8.73%.
- For ASSISTments 2012 the best model is NLP-POST using CountVectorizer embeddings. The model improves the AUC of SAINT+ from 0.736 to 0.793, with an improvement equal to 0.057, corresponding to a percentage improvement of 7.74%.
- For Cloud Academy dataset, the best model is NLP-POST using CountVectorizer embeddings. The model improves the AUC of SAINT+ from 0.759 to 0.771, with an improvement equal to 0.012, corresponding to a percentage improvement of 1.58%.
- For POJ dataset, the best model is NLP-POST using word2vec embeddings. The model improves the AUC of SAINT+ from 0.661 to 0.703, with an improvement equal to 0.042, corresponding to a percentage improvement of 6.35%.

Best among	model	metrics			
		ACC	**AUC**	**Prec**	**Recall**
baselines	DKT skill	0.711	0.721	0.725	**0.896**
models not using text	POST	0.707	0.736	**0.862**	0.725
NLP-enhanced models	NLP-DKT with CountVectorizer	0.735	0.778	0.751	0.876
hybrid approaches	weighted sum of predictions	**0.742**	**0.784**	0.763	0.864

Table 7.17: Results of the best performing models on ASSISTments 2009 dataset.

Best among	model	metrics			
		ACC	**AUC**	**Prec**	**Recall**
baselines	SAINT+	0.707	0.736	0.862	0.725
models not using text	POST	0.754	0.790	0.878	0.784
NLP-enhanced models	NLP-POST with Sentence Transformer	**0.757**	**0.793**	**0.896**	0.779
hybrid approaches	weighted sum of vectors	0.745	0.775	0.762	**0.906**

Table 7.18: Results of the best performing models on ASSISTments 2012 dataset.

Best among	model	metrics			
		ACC	**AUC**	**Prec**	**Recall**
baselines	SAINT+	0.692	0.759	0.721	0.720
models not using text	POST	0.699	0.769	0.695	0.736
NLP-enhanced models	NLP-POST with CountVectorizer	**0.703**	**0.771**	**0.741**	0.725
hybrid approaches	weighted sum of predictions	0.666	0.699	0.669	**0.754**

Table 7.19: Results of the best performing models on Cloud Academy dataset.

Best among	model	metrics			
		ACC	**AUC**	**Prec**	**Recall**
baselines	SAINT+	0.618	0.661	0.557	0.610
models not using text	POST	0.634	0.688	0.647	0.622
NLP-enhanced models	NLP-POST with word2vec	0.644	**0.703**	0.551	**0.651**
hybrid approaches	weighted sum of multiply vectors	**0.653**	0.696	**0.681**	0.631

Table 7.20: Results of the best performing models on POJ dataset.

8 | Conclusion

In this work, we studied multiple directions to improve the Knowledge Tracing task. In particular, we focused on improving the performance on the answer correctness prediction task, evaluating the models on ASSISTments 2009, ASSISTments 2012, Cloud Academy and Peking Online Judge datasets. Since the datasets are unbalanced, models unable to model the knowledge and produce a good prediction tend to predict mostly the majority value, reaching an higher recall, while lowering the precision metric. For this reason, precision and recall are unreliable metrics to evaluate the models and we decided to consider only the Binary Accuracy and the Area Under the ROC Curve to infer conclusions and compare the models.

First of all, we proposed Prediction Oriented Self-Attentive knowledge Tracing (POST), a novel model for KT on the four datasets. Our study proved that POST models the answer correctness prediction task better than the previous self-attention based model SAINT+. In addition, POST has shown the advantage to work optimally even with a small number of training samples.

Then, we examined six Natural Language Processing methods to produce textual embeddings from the text of the exercises and developed "NLP-enhanced" versions of DKT and POST, able to use the textual exercise embeddings as inputs. The significant improvement in the results of NLP-enhanced DKT models compared to DKT proves the utility of "NLP-enhancing" KT models. At the same time, the proposed NLP-enhanced POST model has shown to improve POST both ACC and AUC on all the datasets, with an higher improvement on datasets with few samples. From comparing the six NLP methods (i.e. CountVectorizer, word2vec, doc2vec, DistilBERT, Sentence Transformer and BERTopic), we can conclude that all of them (except BERTopic) can produce good results. However, the results depend too much on the dataset to provide a general ranking of the NLP methods. We can only affirm that BERTopic should not be used to produce exercise embeddings.

Lastly, we presented the possibility of creating "hybrid" models for KT, using at the same time exercise embeddings produced by multiple NLP methods. We propose two possible

approaches to develop a hybrid model, parallelizing different NLP-enhanced DKT models. The improvements given by hybrid NLP-DKT over each dataset prove the capabilities of these models. In addition, a hybrid model provided the highest AUC on *ASSISTments 2009* dataset, improving by 8.73% the AUC of DKT.

To summarize the results, our proposed models increased the AUC metric of previous models by 8.7%, 7.7%, 1.6% and 6.4% respectively on ASSISTments 2009, ASSISTments 2012, Cloud Academy and Peking Online Judge datasets; while the binary accuracy increased by 4.2%, 7.1%, 1.6%, and 5.7% respectively.

Since we accurately investigated the answer prediction correctness task, we can suggest some potential future directions. First, NLP-POST is still unable to use textual embeddings with large datasets, so we suggest replacing its linear layer with another module responsible for reducing the dimension of NLP exercise embeddings. Another possibility is to replace the past exercise content encoder with two encoders: one for skill id and item id embeddings, the other for textual embeddings. In general, we can affirm that a study of the possible ways to use textual embeddings into self-attention is still needed. After having improved NLP-POST, the following step we suggest is to create a hybrid model, combining the embeddings generated by multiple NLP methods. According to the results of hybrid NLP-DKT, performance should receive a large improvement. Due to the wide range of our work, we could not optimize the hyper-parameters of our models or verify how the model works with "cold items", so that is a possible future work. Lastly, other unexplored NLP methods can produce exercise embeddings from text, such as Universal Sentence encoder; studying their utility can be another direction.

Bibliography

[1] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[2] V. Balakrishnan and E. Lloyd-Yemoh. Stemming and lemmatization: A comparison of retrieval performances. 2014.

[3] C. Buciluundefined, R. Caruana, and A. Niculescu-Mizil. Model compression. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, page 535–541, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595933395. doi: 10.1145/1150402.1150464. URL `https://doi.org/10.1145/1150402.1150464`.

[4] Y. Choi, Y. Lee, D. Shin, J. Cho, S. Park, S. Lee, J. Baek, B. Kim, and Y. Jang. Ednet: A large-scale hierarchical dataset in education. *CoRR*, abs/1912.03072, 2019. URL `http://arxiv.org/abs/1912.03072`.

[5] Y. Choi, Y. Lee, J. Cho, J. Baek, B. Kim, Y. Cha, D. Shin, C. Bae, and J. Heo. Towards an appropriate query, key, and value computation for knowledge tracing. *CoRR*, abs/2002.07033, 2020. URL `https://arxiv.org/abs/2002.07033`.

[6] A. T. Corbett and J. R. Anderson. Knowledge tracing: Modeling the acquisition of procedural knowledge. *User Modeling and User-Adapted Interaction*, 4(4):253–278, 1994.

[7] R. Dechter. Learning while searching in constraint-satisfaction problems. 1986.

[8] J. Devlin, M. Chang, K. Lee, and K. Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018. URL `http://arxiv.org/abs/1810.04805`.

[9] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9 (8):1735–1780, 1997.

[10] Z. Huang, Q. Liu, C. Zhai, Y. Yin, E. Chen, W. Gao, and G. Hu. Exploring multi-

objective exercise recommendations in online education systems. pages 1261–1270, 11 2019. ISBN 978-1-4503-6976-3. doi: 10.1145/3357384.3357995.

[11] Q. V. Le and T. Mikolov. Distributed representations of sentences and documents. *CoRR*, abs/1405.4053, 2014. URL `http://arxiv.org/abs/1405.4053`.

[12] E. D. Liddy. Natural language processing. 2001.

[13] Q. Liu, Z. Huang, Y. Yin, E. Chen, H. Xiong, Y. Su, and G. Hu. EKT: exercise-aware knowledge tracing for student performance prediction. *CoRR*, abs/1906.05658, 2019. URL `http://arxiv.org/abs/1906.05658`.

[14] M. Luong, H. Pham, and C. D. Manning. Effective approaches to attention-based neural machine translation. *CoRR*, abs/1508.04025, 2015. URL `http://arxiv.org/abs/1508.04025`.

[15] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *Proceedings of Workshop at ICLR*, 2013, 01 2013.

[16] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. *CoRR*, abs/1310.4546, 2013. URL `http://arxiv.org/abs/1310.4546`.

[17] S. Pandey and G. Karypis. A self-attentive model for knowledge tracing. *CoRR*, abs/1907.06837, 2019. URL `http://arxiv.org/abs/1907.06837`.

[18] S. Pandey and J. Srivastava. RKT : Relation-aware self-attention for knowledge tracing. *CoRR*, abs/2008.12736, 2020. URL `https://arxiv.org/abs/2008.12736`.

[19] C. Piech, J. Spencer, J. Huang, S. Ganguli, M. Sahami, L. J. Guibas, and J. Sohl-Dickstein. Deep knowledge tracing. *CoRR*, abs/1506.05908, 2015. URL `http://arxiv.org/abs/1506.05908`.

[20] N. Reimers and I. Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. *CoRR*, abs/1908.10084, 2019. URL `http://arxiv.org/abs/1908.10084`.

[21] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[22] D. E. Rumelhart and J. L. McClelland. *Learning Internal Representations by Error Propagation*, pages 318–362. 1987.

[23] V. Sanh, L. Debut, J. Chaumond, and T. Wolf. Distilbert, a distilled version of BERT: smaller, faster, cheaper and lighter. *CoRR*, abs/1910.01108, 2019. URL `http://arxiv.org/abs/1910.01108`.

[24] F. Schroff, D. Kalenichenko, and J. Philbin. Facenet: A unified embedding for face recognition and clustering. *CoRR*, abs/1503.03832, 2015. URL http://arxiv.org/abs/1503.03832.

[25] D. Shin, Y. Shim, H. Yu, S. Lee, B. Kim, and Y. Choi. Saint+: Integrating temporal features for ednet correctness prediction. In *LAK21: 11th International Learning Analytics and Knowledge Conference*, LAK21, page 490–496, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450389358. doi: 10.1145/3448139.3448188. URL https://doi.org/10.1145/3448139.3448188.

[26] Y. Su, Q. Liu, Q. Liu, Z. Huang, Y. Yin, E. Chen, C. Ding, S. Wei, and G. Hu. Exercise-enhanced sequential modeling for student performance prediction. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[27] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. *CoRR*, abs/1409.3215, 2014. URL http://arxiv.org/abs/1409.3215.

[28] H. Tong, Y. Zhou, and Z. Wang. Exercise hierarchical feature enhanced knowledge tracing. *CoRR*, abs/2011.09867, 2020. URL https://arxiv.org/abs/2011.09867.

[29] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017. URL http://arxiv.org/abs/1706.03762.

Acknowledgements

Many people supported my effort in this thesis.

I would first like to thank my supervisor, Professor Paolo Cremonesi, for the chance to work on this subject. I would also like to thank my co-advisor, Luca Benedetto, for the constant help and the countless suggestions he has given me during the last year. His passion for the subject made the thesis research very interesting and engaging.

I would like to thank all my friends. In particular, Nicola, Fabio and Giacomo for the shared adventures, which helped to improve my motivation. In the end, I want to dedicate this work to my parents: Giuliano and Silvia, my brother Andrea and my girlfriend Francesca. They are there whenever I need them. Their unconditional love and encouragement always boost me to pursue my dreams and goals.