Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Masterarbeit

Ahmad Khalidi

An Explainability Analysis of BERT's Interpretability with GNN-Generated Knowledge Graph Embeddings Delivered through Soft Prompts

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Ahmad Khalidi

An Explainability Analysis of BERT's Interpretability with GNN-Generated Knowledge Graph Embeddings Delivered through Soft Prompts

Masterarbeit eingereicht im Rahmen der Masterprüfung

im Studiengang Master of Science Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Thomas Clemen
Zweitgutachter: Prof. Dr. Marina Tropmann-Frick

Eingereicht am: 13. März 2025

Ahmad Khalidi

Thema der Arbeit

An Explainability Analysis of BERT's Interpretability with GNN-Generated Knowledge Graph Embeddings Delivered through Soft Prompts

Stichworte

Large Language Modeling, LLM, BERT, Graph Representation Learning, GNN, GCN, Link Prediction, XAI, Explainable AI, Graph Neural Networks, Graph Convolutional Networks, GraphSage, Soft Prompts, GraphPrompter

Kurzzusammenfassung

Trotz großer Fortschritte von großen Sprachmodellen (LLMs) der letzten Jahre in praktischen Anwendungszenarien leiden LLMs an Halluzinationen und Biases. Mit retrieval augmented generation (RAG) werden den Sprachmodellen zur Laufzeit eine Faktenbasis übergeben, auf die sie sich beziehen können. Bei Knowledge Graphen (KG) basiert dieser Prozess auf graph representation learning (GRL), einem Aufgabenfeld, welcher die Strukturen von Knoten und Kanten der Graphen auf niedrigsdimensionale Vektorräume abbildet. Die sogenannten knowledge graph embeddings (KGEs) bilden dabei die strukturelle Einbettung von Knoten und Kanten dieser KGs ab. In dieser Arbeit trainieren wir ein BERT-Modell auf die Klassifikationsaufgabe: *link prediction*. Wir weisen nach, dass sich die Performance steigert, wenn dem BERT-Modell nicht nur die natürlich sprachigen Elemente des KG übergeben werden, sondern vorverarbeitete KGEs, die von graph neuronalen Netzen (GNN) generiert worden sind. Wir untersuchen dann mit Hilfe von explainable AI (XAI) Methoden die inneren Zustände und Verhaltensweisen des BERT-Modells. Es stellte sich dabei heraus, dass BERT die aus dem GNN generierten KGEs nur teilweise zu interpretieren vermag. Dabei unterschied das Modell zwischen verschiedenen Strategien, zwischen denen es anhand weniger offensichtlichen Faktoren entschieden hat. Als eine der wichtigsten Einflussfaktoren stellte sich die Konnektivität von Knoten heraus.

Ahmad Khalidi

Title of the paper

An Explainability Analysis of BERT's Interpretability with GNN-Generated Knowledge Graph Embeddings Delivered through Soft Prompts

Keywords

Large Language Modeling, LLM, BERT, Graph Representation Learning, GNN, GCN, Link

Prediction, XAI, Explainable AI, Graph Neural Networks, Graph Convolutional Networks, GraphSage, Soft Prompts, GraphPrompter

Abstract

Despite major advances in large language models (LLMs) in recent years in practical scenarios, LLMs suffer from hallucinations and biases. With retrieval augmented generation (RAG), the language models are given a fact base at runtime to which they can refer. For knowledge graphs (KG), this process is based on graph representation learning (GRL), a task field that maps the structures of nodes and edges of the graphs to low-dimensional vector spaces. The so-called knowledge graph embeddings (KGEs) represent the structural embedding of nodes and edges of these KGs. In this work, we train a BERT model on the classification task: *link prediction*. We prove that the performance increases when the BERT model is not only given the natural language elements of the KG, but also preprocessed KGEs generated by graph neural networks (GNN). We then use explainable AI (XAI) methods to examine the internal states and behaviors of the BERT model. It turned out that BERT is only partially able to interpret the KGEs generated from the GNN. The model distinguished between different strategies, between which it decided on the basis of less obvious factors. One of the most important influencing factors turned out to be the connectivity of nodes.

Contents

List of Tables

List of Figures

1 Introduction

Large language models (LLMs) such as GPT4 (OpenAI [2023]), Llama3 (Llama Team and AI @ Meta [2024]), Gemma2 (Gemma Team and Google DeepMind [2024]) or Deepseek (DeepSeek-AI [2025]) have shown great success in recent years and promise application in many real-world scenarios. Despite the success of LLMs, critical sources of error, such as **hallucinations**, producing information that is plausible but factually incorrect (Perković et al. [2024], Fernando et al. [2024]) and **biases**, where the prediction distribution may not reflect human values and social contracts and deviates from a subjective distribution (Fernando et al. [2024]), still need to be eliminated. One promising approache to combat hallucinations is **retrieval augmented generation (RAG)**, a technique in which a retriever extracts factual knowledge belonging to a given context from knowledge databases at runtime and makes them available to the LLM (Gao et al. [2023]). The extraction process usually involves mapping the individual sections of the knowledge base to low-dimensional vectors. During extraction, the retriever maps the context to the same dimension and uses a scoring function to search for content that appears similar to the context. The highest scoring content is then passed to the LLM as a knowledge base (Liu [2022]).
Knowledge graphs (KGs) are a special form of knowledge database in which (partially) natural language entities are modeled as nodes and their relationships as edges in a structured graph (Ji et al. [2022]). In **graph representation learning (GRL)** low-dimensional vector representations of nodes, edges or (sub)graphs, sometimes referred to as **knowledge graph embeddings (KGEs)**, are created, which take into account graph properties such as topology and neighborhoods (Khoshraftar and An [2024], Cao et al. [2024]). Combining GRL and RAG is an active field of research and promises to incorporate structural relationships within the graph when selecting relevant nodes (Sophaken et al. [2024], Dong et al. [2024]). The integration of GRL into LLMs is a relevant research focus because LLMs themselves are often used as retrievers in RAG (Karpukhin et al. [2020], Khattab and Zaharia [2020]) and LLMs have difficulties processing structural information from graphs due to their sequential nature in data processing (Khoshraftar and An [2024], Wu et al. [2023]). The reasons mentioned above speak in favor of investigating the inclusion of GRL in LLMs in more detail.

The capabilities of state of the art LLMs are measured as black-boxes by performance in benchmark data sets and human evaluation (OpenAI [2023], Llama Team and AI @ Meta [2024], Gemma Team and Google DeepMind [2024], DeepSeek-AI [2025]). In many areas, however, higher confidence requirements are placed on the behavior of LLMs. It has to be possible to understand how LLMs solve the task assigned to them and which criteria they pay attention to (Mohammadkhani et al. [2023], Kokalj et al. [2021]). The research field of explainable AI (XAI) is concerned with investigating and explaining the behavior of artificial intelligence (AI). The models under consideration, such as LLMs, are difficult to understand due to their complexity and the immense training effort with billions of training data. In order to strengthen trust in the technology and to ensure fairness and possible regulations on responsibility, AI models must be examined for their basic mechanisms in their decision-making and made comprehensible to the user (Kamate et al. [2024], S et al. [2024]). Best to our knowledge there is only one scientific publication using XAI methods to understand the integration of GRL in LLMs in detail (Li et al. [2024]).

The actual learned properties of GRL confronted LLMs have been insufficiently studied in existing research, best to our knowledge. That is why we dedicate this thesis to this research topic. We use a simple example to show how a transformer model reacts to KGEs in the context of a typical GRL task. In doing so, we confirm the assumption that LLMs can benefit from KGEs when performing GRL tasks. Furthermore, we show that one of the biggest influencing factors in the processing of the transformer in our experiments is an unknowingly created artificial bias in the dataset used. We argue that this bias was unnoticed, as it is not relevant from the perspective of GRL, but is relevant in the modality switch to **natural language processing (NLP)**. Building on this, we argue that this kind of effect can be particularly relevant when edges and nodes are passed to LLMs using Graph RAG. There is a lot to be said for passing KGEs to the LLM in addition to the found natural language portions, so that the LLM can also interpret the context of nodes and edges. Our experiments have shown that passing KGEs to the LLM has increased the effect of said bias, which allowed us to identify them more effectively.

Link prediction is a typical GRL task, which aims to predict the source (head entity) s_i or target (tail entity) node t_i of missing triplets $(s_i, r, ?)$ or $(?, r, t_i)$ for given query relation (edge) r (Khoshraftar and An [2024], Cao et al. [2024]). This task can reformulated as a classification task to determine whether an edge exists between two nodes (Khoshraftar and An [2024]). In the experiments in this thesis, we focused on the link prediction task, as it is a GRL task that can

be easily transferred to NLP.
Our implementation of the GRL component is a modification of the link prediction tutorials[1] published by **Pytorch Geometrics (PyG)** (Fey and Lenssen [2019]), in which the authors train a **GraphSage** model (Hamilton et al. [2017]) to create KGEs for the task of link prediction on the **MovieLens** dataset (Harper and Konstan [2015]). This dataset represents a social network in which users rate movies they watched. Because this dataset is large, coming from a real-world scenario and contains natural language node properties, it is well suited for our use case as a representative. GraphSage (Hamilton et al. [2017]) is a well established **Graph Convolutional Network (GCN)** (Ren et al. [2024], Liu et al. [2024a]), that uses local neighborhood sampling to generate node embeddings. The method of local sampling makes GraphSage scalable for large graphs (Bilot et al. [2024]). The PyG link prediction tutorial delivered a well documented, simple and publicly available implementation of a GRL processing pipeline, which is why we chose this pipeline as a starting point for our GRL component.
The LLM of our choice is BERT-Tiny (Turc et al. [2019]). This model is a scaled version of **Bidirectional Encoder Representations from Transformers (BERT)** (Devlin et al. [2019]). We integrate the KGEs into the input vectors of a BERT encoder model (Devlin et al. [2019]) using the GraphPrompter (Liu et al. [2024b]) architecture. In the GraphPrompter architecture, KGEs are concatenated to the input vectors of the LLM via soft prompts (Ng et al. [2024]). We have embedded the optimized implementation in the Python Hugging Face stack (Wolf et al. [2020], Lhoest et al. [2021]). The resulting tool suite can be accessed publicly on our GitHub[2].

[1]Link to the tutorials: https://pytorch-geometric.readthedocs.io/en/latest/get_started/colabs.html
[2]Link to GitHub: https://github.com/AhmadHAW/Hauptprojekt

2 Related Work

2.1 Large Language Model (Transformer)

In recent years the use of transformer architecture, commonly refereed to as LLMs, have shown great success in numerous natural language processing (NLP) tasks, such as question answering, commonsense reasoning, mathematics and science, and coding OpenAI [2023], Gemma Team and Google DeepMind [2024], Llama Team and AI @ Meta [2024]. LLMs are made up of billions of trainable parameters and are trained on massive text datasets (B et al. [2024]). The use of LLMs has already spread into multiple domains like education (Laato et al. [2023]), law (Cheong et al. [2024]), healthcare (Kuzlu et al. [2023], Yang et al. [2024]), finance (Ni et al. [2024]), coding (Fan et al. [2023]) and robotics (Chen et al. [2024a]).

2.1.1 Transformer (Encoder) Architecture

Before the transformer architecture, NLP was largely dominated by recurrent neural networks (RNN), long short term memories (Hochreiter and Schmidhuber [1997]) and gated recurrent neural networks (Chung et al. [2014]). However, the strict sequential processing caused memory issues on longer sequences and models were not able to learn dependencies between distant positions (Vaswani et al. [2017]). In the widely used encoder-decoder architecture, the encoder read an input prompt and mapped it to a vector with a fixed length, which the decoder read and generated the output from (Bahdanau et al. [2014]). This approach also had its difficulties with long input records because the size of the encoder output vector is fixed (Bahdanau et al. [2014], Cho et al. [2014]). The attention mechanism made it possible to learn dependencies between positions in the input and output prompt and thus to focus on specific information within the input prompt while decoding (Niu et al. [2021]).

Vaswani et al. [2017] (first released 2017) were one of the first who got rid of the RNN and realized a model architecture, that fully relies on the attention mechanism instead.

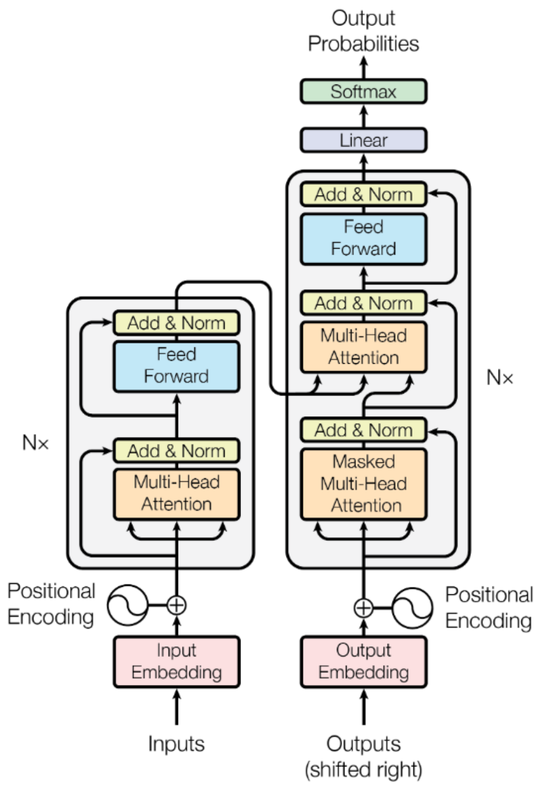


Figure 2.1: The Transformer - model architecture. Reprinted from *Attention is All you Need.*, by Vaswani et al., 2023, Copyright 2023 © by Google. Reprinted with permission.

Figure 2.1 describes their proposed model architecture called "**transformer**". The encoder maps given input sequence to a continues representation space, which is then passed to the decoder to produce the output sequence. The encoder uses 6 stacked layers of multi-head self-attention sub-layer followed by a fully connected sub-layer. Residual layers (He et al. [2016]) are placed around each sub-layer followed by layer normalization (Ba et al. [2016]). The inputs are first transformed to input tokens (**input embedding**) of dimension $d_H = 512$ (hidden size) using a learned weigh matrix. Then the **positional encoding** is added to the input embedding, because the model contains no recurrence or convolution and would not be able to make use of the order of the sequence.

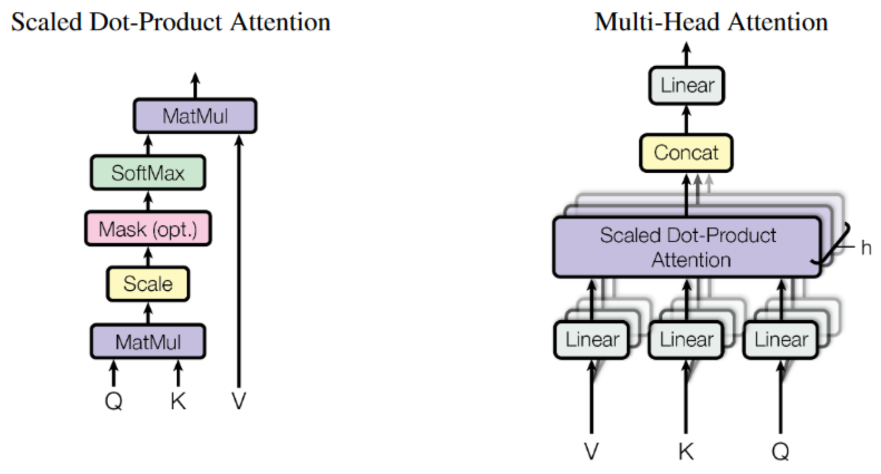


Figure 2.2: Scaled dot-product attention (left). Multi-head attention consists of several attention layers running in parallel (right). Reprinted from *Attention is All you Need.*, by Vaswani et al., 2023, Copyright 2023 © by Google. Reprinted with permission.

Figure 2.2 (left) describes one self-attention head in detail. Each self-attention head computes the weighted sum of a value (V), where the weight is computed by a compatibility function of the query (Q) with the corresponding key (K). The value dimension is d_v and the query and keys dimensions are each d_k. The weights (attention) is computed by the dot product of all keys and the query, then scaled by $\sqrt{d_k}$ and last applying a softmax function. A last dot product is applied to the weights and their values. This attention function is computed on matrices in parallel, resulting in

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

Figure 2.2 (right) shows how instead of performing a single attention function, multiple attention functions are first computed in parallel, concatenated and then linear projected. Also for each parallel self-attention head, a separate linear projection is learned and applied to the inputs for query, keys and values, resulting in
$MultiHead(Q, K, V) = Concat(head_1, ..., head_h)W^O$
where $head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$
where the projections are parameter matrices $W_i^Q \in \mathbb{R}^{d_H \times d_k}$, $W_i^K \in \mathbb{R}^{d_H \times d_k}$, $W_i^V \in \mathbb{R}^{d_H \times d_v}$ and $W^O \in \mathbb{R}^{hd_v \times d_H}$. The authors chose to work with $h = 8$ parallel attention layers, resulting in $d_k = d_v = d_H/h = 64$.

Each multi-head self-attention sub-layers follows a **fully connected feed-forward-network**

(FFN), which is applied to each position separately and identically. The FFN consists of two linear transformations with ReLU activation in between, resulting in:

$$FFN(x) = max(0, xW_1 + b_1)W_2 + b_2$$

The dimensionality of the input layer is $d_H = 512$ and $d_{ff} = 2048$ in all inner layers.
The model was trained on a large English to German translation dataset and resulted in (then) state of the art translation performance.

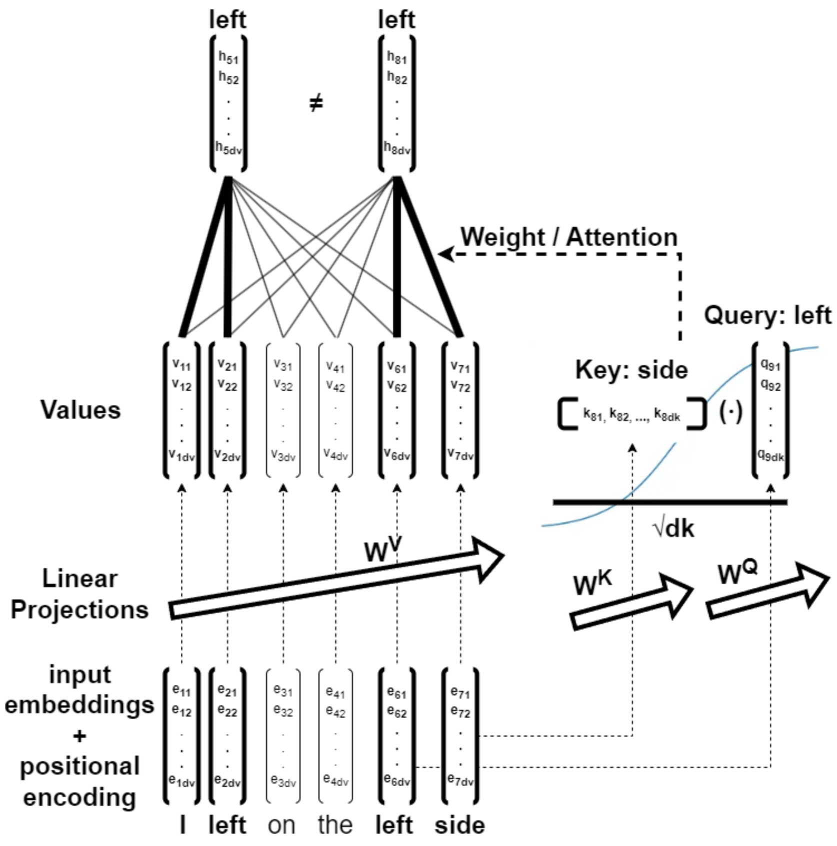


Figure 2.3: The self-attention process as in *Attention is All you Need.*, by Vaswani et al., 2023

Our interpretation of the transformer architecture by Vaswani et al. [2017] illustrated in figure 2.3 illustrates not only the mathematical process of self-attention, but also the semantic meaning behind the vector representations produced. In our example "I left on the left side", we focus

on the meaning of the homographic words[1] “left” within the vector representation. First the encoded words[2] are transformed into values, keys and queries with the linear projections. We can assume that the values of the words "left" (v_2 and v_5) are similar to each other at this point. The main difference in their semantic meaning is computed by the self-attention process. Here the first position of the word "left" (h_1) is paying high attention to the word "I", which indicates, that the word "left" can be interpreted as the conjunction of the word "leaving". The attention process is illustrated in detail on the second position of the word "left. Here we assume, that the position is paying high attention to the word "side", because it indicates that "left" can be interpreted as a direction. If the attention (weight) for this position is great, then the dot product of key and value must have been great and this can only be the case, if the similarity between key and value vectors were great. In other words: The transformer learns to produce the linear projections of words to be similar, if they are highly contextual dependent. Here the only difference of the query vectors of "left" were the positional encodings, that were added before. These encodings made sure, that the relative positions of "I" and "side" were taken into consideration.
The resulting vector representations of all attention heads are then concatenated and passed into a FFN for each position. So the FFN takes all *views* of contextual dependencies and generates vector representations of the words in their context. Here we can assume, that the word "left" will be represented as the verb in the first position and as the direction in the second position. We can also assume that for more complex and deeper levels of transformer architectures, the vector representations will become semantically richer.

2.1.2 BERT

Pre-Training LLMs has shown great success in improving the performance of many specific downstream tasks (Peters et al. [2018], Radford and Narasimhan [2018]). The authors of textbf-BERT (Devlin et al. [2019]) applied two pre-training techniques to the transformer architecture (Vaswani et al. [2017]). They have shown, that the encoder part of the transformer can be taught to produce semantically rich embeddings of text tokens only by pre-training on a large unlabeled textual corpus. These pre-trained **foundation models** are then applicable to specific downstream tasks by little modifications on the model architecture and relatively little effort in finetuning.
The first pre-training task applied on large corpus of text data was **masked language modeling (MLM)**, a training technique inspired by "cloze task" (Taylor [1953]), in which text tokens are randomly masked in the input sequence by replacing them with the specialized token *[MASK]*.

[1]Words that have the same spelling but different meaning.

[2]This is a simplification of the encoding process. Here we assumed that each word is encoded as one vector, though in reality longer words will be represented by multiple vectors.

The transformer model was then trained on predicting the masked token. For that, the output vector representations of the masked token positions are passed through a softmax layer.
The second pre-training task applied to BERT was **next sentence prediction (NSP)**. The authors divided the text corpus into sentence pairs, where each pair is a consecutive sentence pair in the text corpus. Then they switched 50% of the second sentences with random sentences and made BERT predict, if the second sentence actually follows the first sentence. For that, they introduced two additional special tokens. The *[SEP]* token is used to separate the sentence pair in the token sequence and to mark the end of the token sequence. The token *[CLS]* is added at the beginning of the token sequence and is used for classification tasks, by only passing its output vector representation to an output layer for classification.
The authors also added **segment embeddings** to the input embeddings in addition to the positional encoding. These embeddings allow the LLM to separate the sentences in next sentence prediction even better. The positional encodings became **positional embeddings** by replacing the encoding function with a trainable linear projection. The authors also added more layers of multi-head attention blocks to the architecture. They increased the number of parallel attention heads and the size of the internal vector representation.
BERT was trained on top of the **WordPiece** embeddings method (Wu et al. [2016]) who's behavior has been thoroughly investigated in scientific research (Clark et al. [2019], Rogers et al. [2020], Tenney et al. [2019], Jiang et al. [2021], Hewitt and Manning [2019], Goldberg [2019], Coenen et al. [2019]). We illustrate the MLM pre-training procedure in BERT (Devlin et al. [2019]) using figure 2.4. In this example, the word "is" is masked at token ID level with the *[MASK]* token. In addition to the positions, a segment is assigned to each token. In this case, each token has the same segment. Token IDs, positions and segments are then mapped to their embedding using FFNs, which are then added together in the next step. The main body of BERT corresponds to the transformer architecture as introduced by Vaswani et al. [2017], but with the encoder only part and the corresponding size changes. The encoder produces output vectors for all tokens, including the mask token. We can assume that the mask token in the output layer combines the semantic context in which it is located and thus carries enough information to infer the masked token. The resulting embedding of size 786 is then scaled up in a linear projection in a likelihood over the token dictionary and then passed into a softmax, resulting in a ranked order of tokens to be placed into the masked token position. This training teaches the model to produce semantic contextual embeddings at each position.

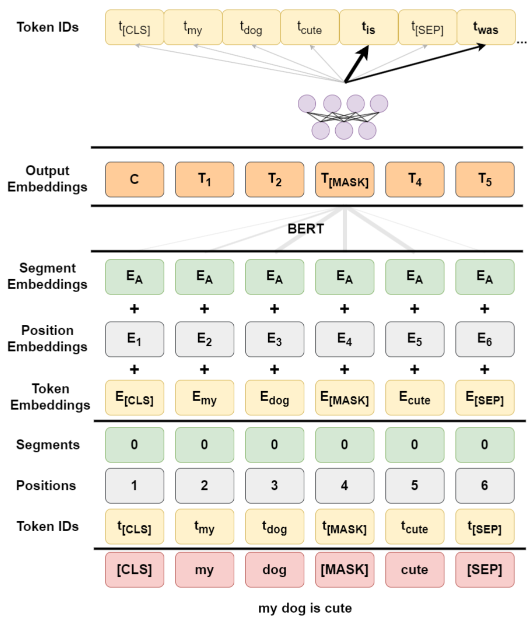


Figure 2.4: Masked language modeling in BERT illustration. Adapted from *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*, by Devlin et al., 2019

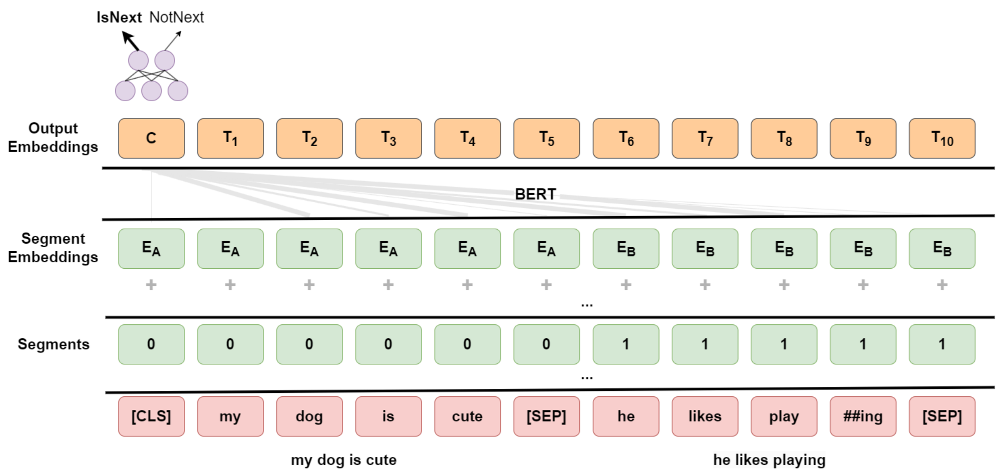

IsNext NotNext
Output Embeddings
C T1 T2 T3 T4 T5 T6 T7 T8 T9 T10
BERT
Segment Embeddings
EA EA EA EA EA EA EB EB EB EB EB
Segments
0 0 0 0 0 0 1 1 1 1 1
[CLS] my dog is cute [SEP] he likes play ##ing [SEP]
my dog is cute
he likes playing

Figure 2.5: Next sentence prediction in BERT illustration. Adapted from *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*, by Devlin et al., 2019

Figure 2.5 illustrates the NSP process. This time, each sentence is assigned to a segment 0 or 1. Their segment embeddings are also adapted accordingly. The output embeddings of the

classification token (C) summarize the relationship between both sentences and is passed to a linear projection an scaled down to the classification outputs *IsNext* or *NotNext*. Again, the output is passed into a softmax to find the classes with highest likelihood.
We can summarize, that BERT is trained on the tasks MLM and NSP. MLM teaches the model in generating semantic contextual embeddings of singular tokens, while NSP trains the model to generate semantic contextual embeddings of token groups.

Bert-Tiny (Turc et al. [2019]) is a compact version of BERT (Devlin et al. [2019]) using the principles of **knowledge distillation (KD)** (Hinton et al. [2015]) drastically reduces the parameter size without significantly reducing performance. The set of 24 BERT model miniatures released by Turc et al. [2019] are all based on the transformer architecture (Vaswani et al. [2017]) and trained similar but not equally as BERT (Devlin et al. [2019]). Given the larger teacher model BERT (Devlin et al. [2019]) and the compact student model BERT-Tiny the student model is trained on the soft labels of the teacher model after the extensive pre-training. Soft labels are the output distributions for any given forward pass. The pre-training makes sure, that the student model learns to produce meaningful embeddings in the first place and then the student learns to imitate the confidence distribution of the teacher, even on unseen data (Turc et al. [2019]).

2.2 Graph Representation Learning

KGs became more prominent in NLP, in the hope of providing LLMs with a comprehensible knowledge base to refer to (Abu-Rasheed et al. [2024]) and enable them to process complex relationship structures between entities (Liang et al. [2024]). Processing KGs often includes GRL, a fundamental task that aims to encode high-dimensional graph structures into low-dimensional vector representations (Ju et al. [2024], Khoshraftar and An [2024]). Because KGs encode structured factual knowledge with multi-relational edges (Cao et al. [2024]), multi-modal nodes (Sun et al. [2020]) and follow logical rules, constraints and ontologies, we call the low-dimensional vector representations of KGs **Knowledge Graph Embeddings (KGEs)** (Cao et al. [2024]).

2.2.1 Knowledge Graphs

Knowledge Graphs (KGs) are structured data where entities are in relation with each other and facts (datapoints) are denoted as a triple *(h,r,t)*, where *h* represents the head (source node), *r* the relation (edge) between nodes and *t* the tail (target node) (Ji et al. [2022]). A knowledge graph can store and represent factual knowledge, like semantic networks (Miller [1995]), common sense (Ilievski et al. [2021]), social networks (Alonso et al. [2019], Harper and Konstan [2015]), academic networks (Tang et al. [2008], Sinha et al. [2015]), molecule graphs (Wu et al. [2018]) or traffic networks (Cui et al. [2019]).

Harper and Konstan [2015] provided a rating data set from the MovieLens web service[3], that has been collected and made available by GroupLens Research. The dataset provides 32 million ratings and two million tag applications applied to 87,585 movies by 200,948 independent users over time, collected in 2023 and released 2024.
Each user is represented by artificial integer ids. Each movies is represented by an artificial integer id, a title (with release year), a list of genres and tags. The tags were provided and rated by the user as well. Each rating of a movie by a user is saved with its timestamp. Rating a movie results in a fact of shape *(user,rating,movie,timestamp)*. This triggers the web service recommends other movies, based on the users history (facts).

2.2.2 Link Prediction

One of many GRL downstream tasks is link prediction, with the goal of identify missing entities in fact triplets like *(h,r,?)* or *(?,r,t)* (Akrami et al. [2018]). Link prediction algorithms are often based on node similarity, where the similarity score between two nodes defines the likelihood of an edge between them (Xu and Yin [2017]).
The similarity for KGs can be defined in many ways, depending on the actual downstream task. A common similarity factor is **structural similarity**, where nodes that share the same neighborhood or common connectivity patterns are considered similar (Yu et al. [2024]). The similarity between two vectors can be measured with the dot product between those vectors (Cao et al. [2024]). A popular method of comparing sentence embeddings is cosine similarity, a normalized version of the dot product between two vectors (Reimers and Gurevych [2019]). The combination of both the structural view and a natural language view is an ongoing research field with promising outcomes (Dong et al. [2024]).

2.2.3 Graph Neural Networks

A **graph neural networks (GNN)** is based on information diffusion, where a set of units exchanges information in a graph structure, representing each node individually. These units update their state according to the graph structure until they reach a unique stable equilibrium (Scarselli et al. [2009]).

[3]https://movielens.org

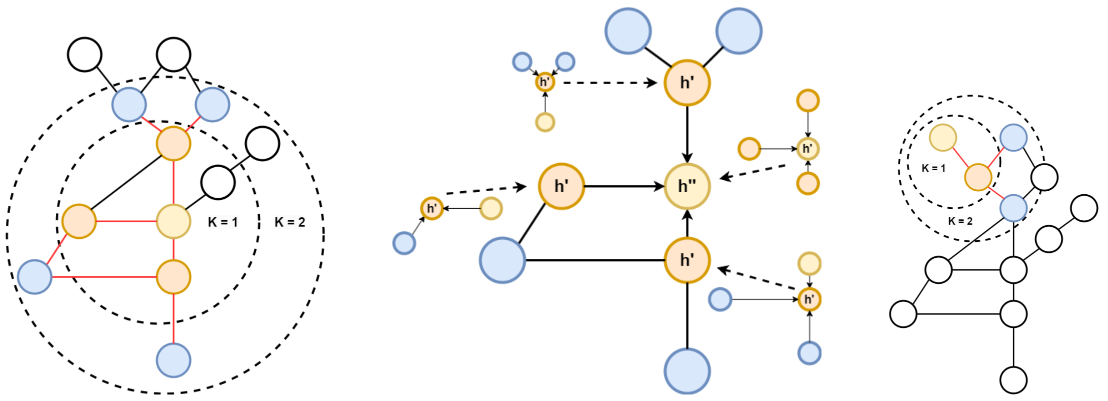
The **Graph convolutional network (GCN)** is based on GNNs and a variant of a convolutional neural network (CNN). Instead of a kernel sliding over pixels, a node aggregates features from its neighbors (Kipf and Welling [2016]).
One common concept for aggregating node embeddings is the **message passing concept**: Given a node *i* at each layer *l + 1*, the node representations of *i* are updated with the following formula:

$$h_i^{l+1} = f(h_i^l, \sum_{j \in N(i)} g(i, j))$$

where f and g are learnable functions, $N(i)$ are the neighbors of node i and h_i^0 are the node i initial features. The GNN aggregates the embeddings of the neighbor nodes at each layer with h_i^L the final representation of node i aggregated with neighbors L hops away (Khoshraftar and An [2024], Liang [2023]).
GraphSage (Hamilton et al. [2017]) is one interpretation of this message passing concept and implements the algorithm the following way:

$$h_i^{'} = W_1 h_i + W_2 * mean_{j \in N(i)} h_j$$

where $h_i^{'}$ is the nodes features after aggregating over the sample neighborhood $N(i)$ with the trainable weight matrices W_1 and W_2. GraphSage encourages nodes with edges to be similar while enforces the representations of disparate nodes to be highly distinct (Hamilton et al. [2017]).
Figures 2.6 illustrate the GraphSage representation learning process as described in Hamilton et al. [2017]. On figure 2.6a we can see the process of sampling nodes over 2 hops where the amount of neighbors sampled in the first hop is 3 and the amount of neighbors sampled in the second hop is 2. In this example we sampled around the central node (yellow). As we can see, not all neighbor nodes of yellow are sampled. Each edge that was traversed during that hop (red) is also sampled. The second hop has multiple effects illustrated. The orange node in the bottom samples two neighbor nodes in the second hop (blue). The orange node on the left samples only one new neighbor node (blue), because it is possible that the edge that was just traversed in the first hop is traversed back, counting as one sample. With the second sample to the bottom left blue node, the neighborhood samples for the left orange node is at it's limit. The orange node at the top also samples 2 new blue nodes. That resulted in the edge between the two top orange nodes not to be sampled in the sampled neighborhood.
Figure 2.6b illustrates the aggregation process of nodes inside the sampled neighborhood. First, each node initializes it's vector representation h_i. Then each vector representation is multiplied



(a) GraphSage sampling first hop 3 neighbors, second hop 2 neighbors

(b) GraphSage convolutional layers aggregate node representations

(c) GraphSage border nodes

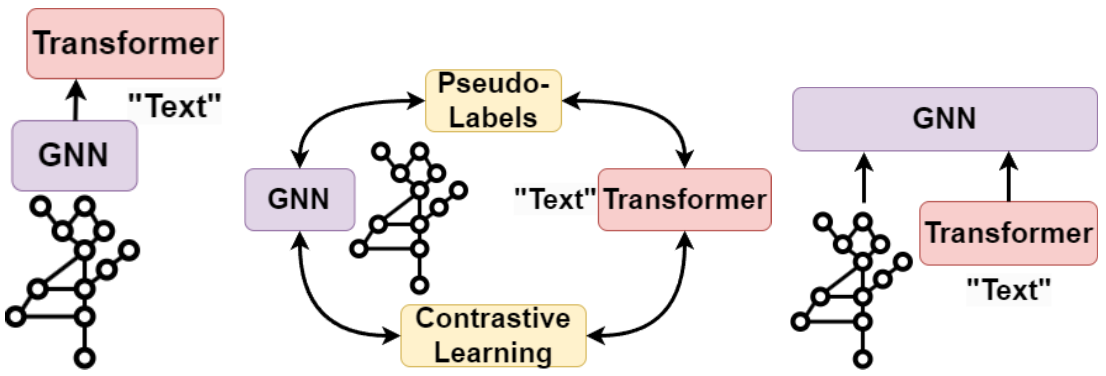
Figure 2.6: GraphSage sampling and aggregation process. Adapted from *Inductive Representation Learning on Large Graphs*, by Hamilton et al., 2017

by the learned weight matrices W_1 and W_2. For each node the vector representations h'_i are computed by adding the weighted vector representations to each other, while the weighted vectors of local neighbor's nodes are averaged. The smaller sub-graphs represent the local neighborhood of each node. In the first hop, the yellow node considers the vector representation of it's direct neighbors (orange). In the second hop, h''_i is computed by aggregating the weighted vector representations of the (already) aggregated neighborhood nodes h'_i (orange). These aggregations already contain the local neighborhood's information. This way the blue node's vector representations indirectly influence the yellow node's vector local neighborhood structural vector representation.
Figure 2.6c illustrates the local neighborhood effect on nodes, that are less connected and can be considered at the border of a cluster. These nodes will have a chance (depending on the graphs structure) to learn a vector representation, that relates them to the rest of the cluster. This behavior will become more important, if we compare them to transformer models processing graph structures on a textual semantic view.

2.3 Transformer in GRL

There are multiple arguments to consider transformer models self-attention mechanism (Vaswani et al. [2017]) as a special procedure of fully connected implicit GRL (Wu et al. [2023]). Transformers show great potential in GRL, especially in the processing of text-based KGs. However, the widely used pre-trained transformer models are said to have many disadvantages when dealing with graphs. On the one hand, most transformer models are not pre-trained in GRL tasks, which reduces their understanding of graph structures even after fine-tuning. In addition, the processing of large graphs does not scale if the graph structures are passed as prompt input (Chen et al. [2024b]). Using LLMs in GRL will therefore require intensive changes to the architecture and/or pre-training.

There are multiple strategies of mixing GNN architectures with transformer architectures. There are graph transformers (Yun et al. [2020]), who use the attention mechanism in their GNN architecture. There are also three other strategies in the cooperation of GNN and transformer models: *transformer as predictor and GNN as encoder* (Jin et al. [2024], Chen et al. [2024b], Liu et al. [2024b]), *transformer and GNN alignment* Jin et al. [2024], Chen et al. [2024b], Tang et al. [2024], and *transformer as encoder and GNN as predictor* (Jin et al. [2024], Chen et al. [2024b]).



(a) Transformer as predictor and GNN as encoder. (b) Transformer and GNN alignment with either pseudo-labels or contrastive learning. (c) Transformer as encoder and GNN as predictor.

Figure 2.7: Strategies fusing the strengths of transformer models and GNNs. Adapted from *Large Language Models on Graphs: A Comprehensive Survey*, by Jin et al., 2024

Figure 2.7 illustrates three strategies that combine the strengths of GNN's GRL processing and transformers NLP. Figure 2.7a illustrates the first strategy, in which the transformer is the predictor and the GNN is encoding the graph structure via GRL. This encoding is then passed to the transformer in combination with the natural texts of the graph for processing (Jin et al. [2024], Liu et al. [2024b]). Figure 2.7b illustrates two strategies for aligning GRL with NLP via either Pseudo-Labels, that the transformer and GNN generate for each other or via contrastive learning, in which the respective encodings are aligned with each other (Jin et al. [2024], Tang et al. [2024]). The last figure 2.7c illustrates the strategy of placing the GNN as the predictor and the transformer encoding the text via NLP (Jin et al. [2024], Chen et al. [2024b]).

Liu et al. [2024b] provide an architecture for the *transformer as predictor* interaction, using the soft prompt approach.
Soft prompts belong to the family of prompt tuning methods, which allow LLMs to be trained on multiple downstream tasks without excessive training. Instead, the main models parameters are being frozen and for each downstream task a separate set of embeddings is randomly initialized. Only these separate embeddings and their parameters are trained on labeled data (Lester et al. [2021]).

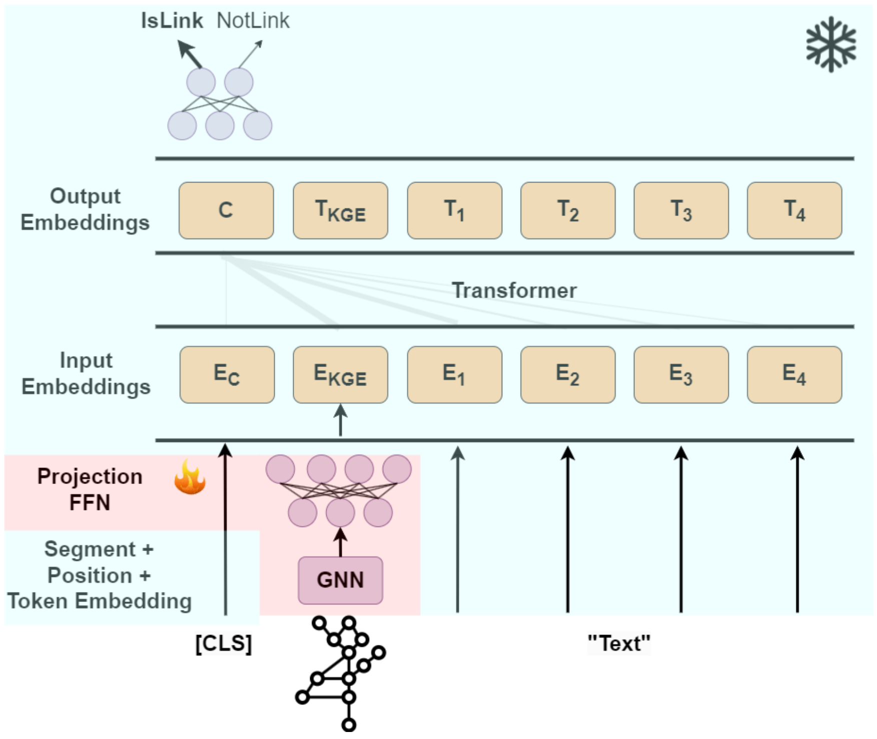


Figure 2.8: GraphPrompter architecture on the task of link prediction adapted from *Can we Soft Prompt LLMs for Graph Learning Tasks?*, by Lio et al., 2024 and *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*, by Devlin et al., 2019

Liu et al. [2024b] uses the same approach in their **GraphPrompter** architecture as depicted in 2.8, but instead of initializing random embeddings for each downstream task, the KGEs of the

GNN are passed to the transformer instead. The whole setup is trained end-to-end, while the transformer's parameter are kept frozen. Every KGE is concatenated into the input embeddings, which are then processed by the transformer for the downstream task link-prediction.

2.4 Explainable AI on LLMs

AI is becoming an increasingly important part of our daily lives. High-performance deep neural networks, such as LLMs, are becoming increasingly complex and less transparent (Singh et al. [2024], Volkov and Averkin [2024], Zhao et al. [2024]). In order to strengthen trust in this technology, ensure fairness and comply with regulations, experts and laypeople must be able to understand and explain the underlying mechanisms and decision-making processes of AI models (Zhao et al. [2024]). LLMs in particular often struggle with hallucinations (Zhao et al. [2024], Volkov and Averkin [2024]) and biases (Fernando et al. [2024]). The scientific community has not yet agreed on a fixed definition for **explainable AI (XAI)**, but we can assume that XAI is a set of techniques that achieve the above-mentioned goals of explainability (Speith [2022]),[Tropmann-Frick et al., 2024, Chapter 3]. Most XAI methods can be categorized into a **local or global scope**, where a model is explained locally a single prediction or globally the whole model (Speith [2022]),[Tropmann-Frick et al., 2024, Chapter 3] and **model-specific or model-agnostic**, where model-specific XAI methods are designed for specific models and model-agnostic XAI methods can be applied to all models (Speith [2022], Zhang et al. [2024]),[Tropmann-Frick et al., 2024, Chapter 3].
XAI methods for LLMs can further be labeled as **classifier-based probing**, where a shallow classifier is trained on top of an LLM that outputs certain linguistic properties and reasoning abilities. Then **vector representations** of the LLM are studied to measure the embedded **syntax and semantic knowledge** Zhao et al. [2024], Volkov and Averkin [2024]. Another possible XAI methods on LLMs are **concept-based explanations**, where the input texts are mapped to a set of concepts and the importance for each pre-defined concept can be measured (Zhao et al. [2024], Mohammadkhani et al. [2023]).

2.4.1 Attention Based

The most common, intuitive, model-specific and local explanation methods are attention based. This approach leverages the LLM's multi head attention layer and analyzes the attention weights between positions in between layers (Mohammadkhani et al. [2023], Vig [2019]). These **attention maps** can be visualized in bipartite graphs (Zhao et al. [2024], Vig [2019]) or heat-maps (Zhao et al. [2024], Mohammadkhani et al. [2023]).

In Mohammadkhani et al. [2023] the authors conducted an empirical study to analyze code models on three domain specific downstream tasks. For that they mixed the local attention based method with concept-based explanations. They grouped input tokens of coding tasks into semantic meaningful groups, like *method name, input variables ....* The resulting attention scores gave them global insights over how certain semantic groups affect the models prediction the most.

2.4.2 SHAP

SHapley Additive exPlanations (SHAP) (Lundberg and Lee [2017]) is a popular model-agnostic and local XAI method (Volkov and Averkin [2024], Zhao et al. [2024]). SHAP (Lundberg and Lee [2017]) treats features of the input sequence as players in a cooperative prediction game. The **SHAP-value** represents the *marginal contribution* of a feature with and without the feature in every possible combination of other features. The contribution of a feature is the average over all possible subsets of other features.
SHAP-value for a given feature i is calculated as follows:

$$\varphi_i = \sum_{S \subset N \setminus \{i\}} \frac{|S|!(|N| - |S| - 1)!}{|N|!} [f(S \cup \{i\}) - f(S)] \tag{2.1}$$

, where N is the set of all features, S is a subset of features excluding i, $f(S)$ is the is the model prediction using the features in subset S and the term $\frac{|S|!(|N|-|S|-1)!}{|N|!}$ is a weighting factor that accounts for the number of subsets.
Computing the SHAP-values for every feature is computational very expensive, because the outcome of every subset over every feature has to be computed, which grows exponentially with the amount of features (Volkov and Averkin [2024], Zhao et al. [2024]).

Because LLMs can be very resource intensive and work on token-level prediction, calculating the SHAP-value is not sufficient (Volkov and Averkin [2024], Zhao et al. [2024]). **TransSHAP** (Kokalj et al. [2021]) is adapting SHAP to the transformer architecture, making it a viable tool. TransSHAP is grouping tokens on higher hierarchical levels based on their importance or similarity. This reduces the need to compute the SHAP-value for every token separately. TransSHAP also leverages the attention mechanism and reduces the amount of subsets created in the process, by sampling only subsets of features that are more relevant according to the attention weights. TransSHAP is generating all possible subsets of features, by masking tokens with the BERT (Devlin et al. [2019]) mask token (*[MASK]*), which is more consistent with the transformer's inner workings, making the process more efficient in computation. Compared to the traditional SHAP-value (Lundberg and Lee [2017]), the TransSHAP-value is less accurate and more probabilistic, but more efficient to calculate (Kokalj et al. [2021]).

2.4.3 PCA on Representation Probing

Another common approach to analyze the semantic knowledge embedded in the internal vector representations of an LLM is by probing the model on a specific downstream task and analyze the internal vector representation with some form of dimension reduction (Jawahar et al. [2019], Coenen et al. [2019], Singh et al. [2024]). **Principal component analysis (PCA)** (Pearson [1901], Hotelling [1933]) is a linear dimensionality reduction technique that seeks to find hyperplanes in lower dimensions, which capture the most variance of the original data. Mathematically, this is done by finding the eigenvector of the covariance matrix with the highest (second-highest, ...) eigenvalue.

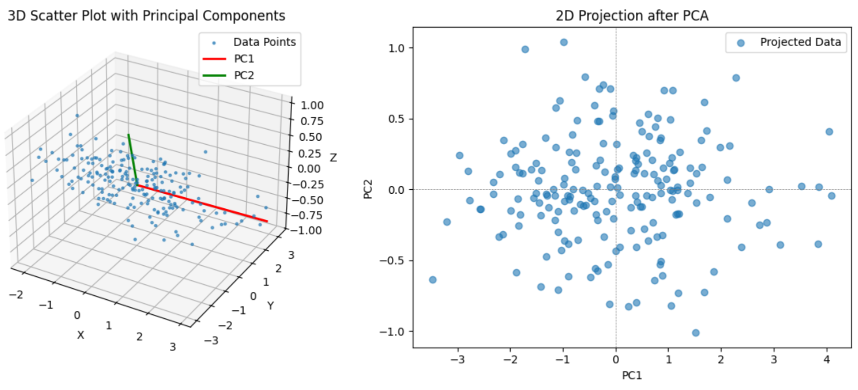


Figure 2.9: Illustration of principal component Analysis from 3 dimensions (left) to the 2 first principal components (right).

Figure 2.9 shows the first two principal components (red and green), that capture the highest variance of the original data points.
This technique can be used to reduce the high-dimensional hidden states of BERT in order to analyze them (Coenen et al. [2019]).

2.5 Open Source Python Libraries

Python (Van Rossum and Drake [2009]) is a programming language that provides a large ecosystem and community when it comes to AI development and machine learning. Python supports an official package index with third-party python software like Pytorch (Paszke et al.

[2019]), Numpy (Harris et al. [2020]), Hugging Face transformers (Wolf et al. [2020]), Hugging Face datasets (Lhoest et al. [2021]) and Pytorch Geometrics (Fey and Lenssen [2019]).

2.5.1 PyTorch

PyTorch is: "a Python library that performs immediate execution of dynamic tensor computations with automatic differentiation and GPU acceleration, and does so while maintaining performance comparable to the fastest current libraries for deep learning." (Paszke et al. [2019]). PyTorch automatically catches differentiable **gradients** when performing operations on **tensors**, which allow convenient back-propagation when training models.

The class ***torch.nn.Module*** is the base class for all neural network and other parameterized modules in PyTorch. It provides methods for defining layers and operations, managing model parameters, and handling forward and backward passes. Every subclass of the nn.Module has to implement the two key methods *__init__()*: The models layers and other components and *forward()*: the forward pass for the data through the model.

The class ***torch.nn.Linear*** inherits from *torch.nn.Module* and applies a linear transformation on incoming tensors using $y = xW^T + b$, where x is the input tensor of shape H_{in}, y is the output tensor of shape H_{out}, W is the trainable weight matrix of shape (H_{out}, H_{in}) and b is the trainable bias of shape H_{out}. This shapes are set fixed during initialization process.

The class ***torch.nn.Embedding*** also inherits from *torch.nn.Module* and is a trainable lookup table that stores embeddings of fixed dictionary size. Given a list of indices, the module returns an embedding of shape H_{embed} for each index in the list, where each embedding is trainable. The module is initialized with the dictionary size and the embedding shape.

2.5.2 Numpy

Numpy (Harris et al. [2020]) is an open source python library for high performance matrix and high dimensional array manipulation on CPU. Numpy plays an essential role in many research analysis pipelines and give the foundation for other scientific open source Python libraries. Each Numpy array comes with metadata, like shape or type. Torch and Numpy tensors are interchangeable on CPU.

2.5.3 Pandas

Pandas (Mc Kinney [2010]) is a python library that handles in-memory tabular data sets and provides basic statistical tools. Pandas implements the ***DataFrame*** class, that helps to integrate structured *Numpy* arrays with the rest if *Numpy*. Multiple functionalities are integrated, such as loading and saving *DataFrames* from *csv*, filtering, grouping, joining multiple *DataFrames* or adding rows or columns to existing ones.

2.5.4 Hugging Face Stack

Hugging Face, Inc. is an US-American Software Company that provides tools for developing machine learning applications, mostly transformers. Hugging Face provides a publicly accessible model and dataset hub and the open source **transformers** and *datasets* python libraries under the Apache 2.0 licence.

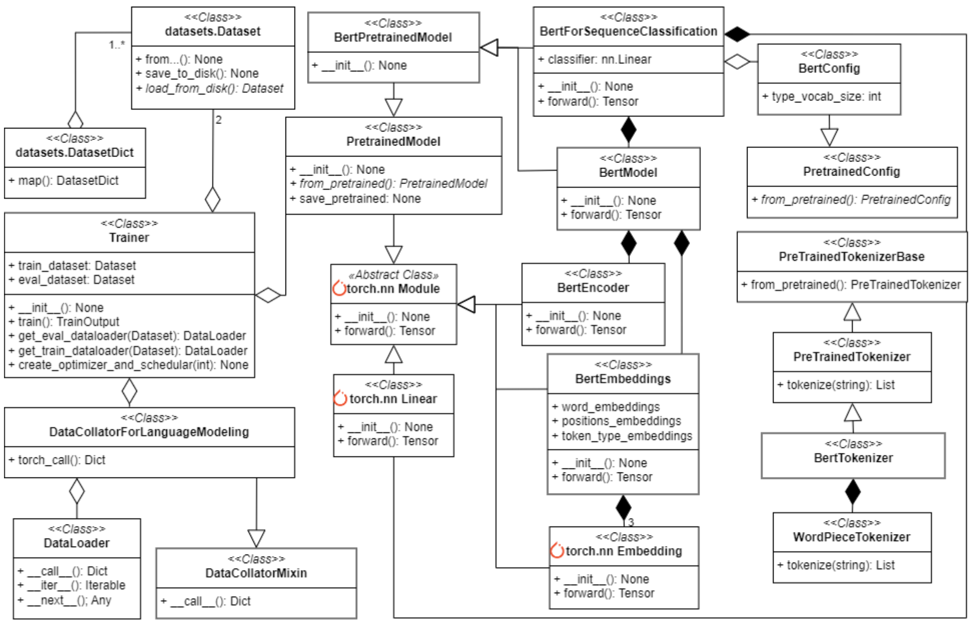
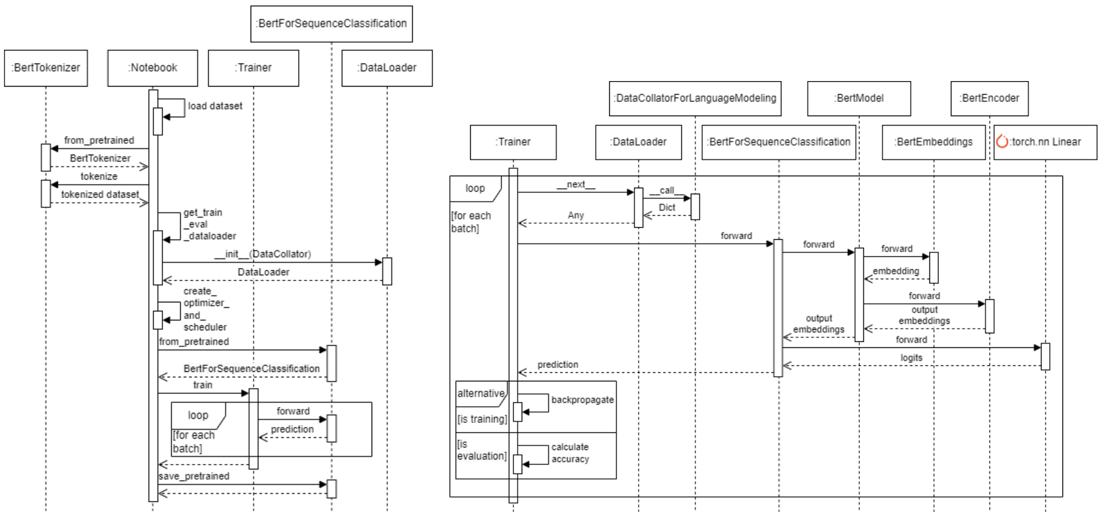


Figure 2.10: A condensed UML class diagram view on the Hugging Face stack of transformers and datasets as in *Transformers: State-of-the-Art Natural Language Processing*, by Wolf et al., 2020 and *Datasets: A Community Library for Natural Language Processing*, by Lhoest et al., 2021

Figure 2.10 shows a condensed UML class diagram overview over the Hugging Face transformers and datasets stack[4]. If there is a preamble before the class name, that preamble indicates the python library and package of the class (for example *datasets.Dataset*). No preamble indicates this class belongs to the Hugging Face *Transformers* library (for example *BertModel*). The PyTorch logo is added to all PyTorch libraries for clarification (for example *torch.nn Module*).



(a) The overall Training process.

(b) The detailed training process inside each batch.

Figure 2.11: The Hugging Face BERT transformer training process as UML sequence diagram: overall and in detail, as in *Transformers: State-of-the-Art Natural Language Processing*, by Wolf et al., 2020.

Figure 2.11 illustrates a typical training pipeline using the Hugging Face transformers library (Wolf et al. [2020]). Figure 2.11a illustrates the preprocessing steps in more detail, while figure 2.11b illustrates the training process of each batch in more detail. In the following we describe all classes from figure 2.10 and then describe their interaction during training as shown in figure 2.11.

Datasets (Lhoest et al. [2021]) is an open source, standardized interface library for NLP datasets. Datasets are tabular, versioned and can be downloaded with little effort. They are computation- and memory-efficient and allow working seamlessly with tokenization. A dataset is represented by ***datasets.Dataset***, which again is an arrow table. Datasets can be loaded from multiple sources, including disk or *Pandas DataFrames*. Datasets can be aggregated by ***datasets.DatasetDict***,

[4]Disclaimer: We are showing the classes only in partial and focus on the classes, methods and parameters that we have used in this project.

which maps given unique strings to *datasets.Dataset* objects.

Transformers library[5] (Wolf et al. [2020]) is based on PyTorch and provides a standard NLP machine learning model pipeline to process data, apply models and make predictions. A complete Hugging Face model is defined by three building blocks: **a tokenizer**, that converts input texts to numeric encodings (***input ids***) and back, a transformer, which transforms the numeric encodings into contextual (internal) embeddings (***hidden states***) and a **head**, which uses the contextual output embedding for the specific downstream tasks, like classification.
The base class of each model is ***PretrainedModel***. *PretrainedModel* does inherit from *torch.nn.Module* but does not implement a *forward* method. That means this class is not trainable but it provides all inheriting classes methods to load and store trained parameters using *from_pretrained* and *save_pretrained*.
Hugging Face implements the BERT models[6] (Devlin et al. [2019]) and (Turc et al. [2019]). ***BertPretrainedModel*** inherits from *PretrainedModel* and handles weight initialization and provides a simple interface for loading pretrained BERT models. ***BertModel*** inherits from *BertPretrained* and implements the actual BERT architecture, as described in (Devlin et al. [2019]) and (Turc et al. [2019]). Each *BertModel* can be initialized with a ***BertConfig***, which inherits from ***PretrainedConfig***, can be loaded from a pretrained state and sets models parameters, like the segments set(here *type_vocab_size*). The *BertModel* also holds reference to ***BertEmbeddings***, a class that provides three *torch.nn.Embedding*s for the initial encoding of tokens (*word_embeddings*), positions (*positions_embeddings*) and segments (*token_type_embeddings*). A forward pass with a list of indices (*input ids*, positions and segments) looks up the respective embeddings and sums them for over each index as described in Devlin et al. [2019].
The other *torch.nn.Module BertModel* holds a reference to is ***BertEncoder***, which implements the actual *forward* pass into the multi-head attention blocks, as described in Devlin et al. [2019]. The outputs of ***BertEncoder*** can be interpreted by a wrapping class ***BertForSequenceClassification***, which passes the output embedding of the classification token (pooled output) of the *BertModel* into a classification (linear) layer.
Hugging Face provides a feature-complete training and evaluation processing for Hugging Face transformers and datasets with the ***Trainer*** class. A trainer holds reference to a training and evaluation dataset. The referenced ***DataCollatorForLanguageModeling*** inherits from ***DataCollatorMixin*** and is responsible for batching incoming lists of datapoints. The trainer initializes a ***DataLoader*** with the *DataCollatorForLanguageModeling* and the training and evaluation

[5]Latest implementation details: https://huggingface.co/docs/transformers/index
[6]https://github.com/huggingface/transformers/blob/main/src/transformers/models/bert/modeling_bert.py

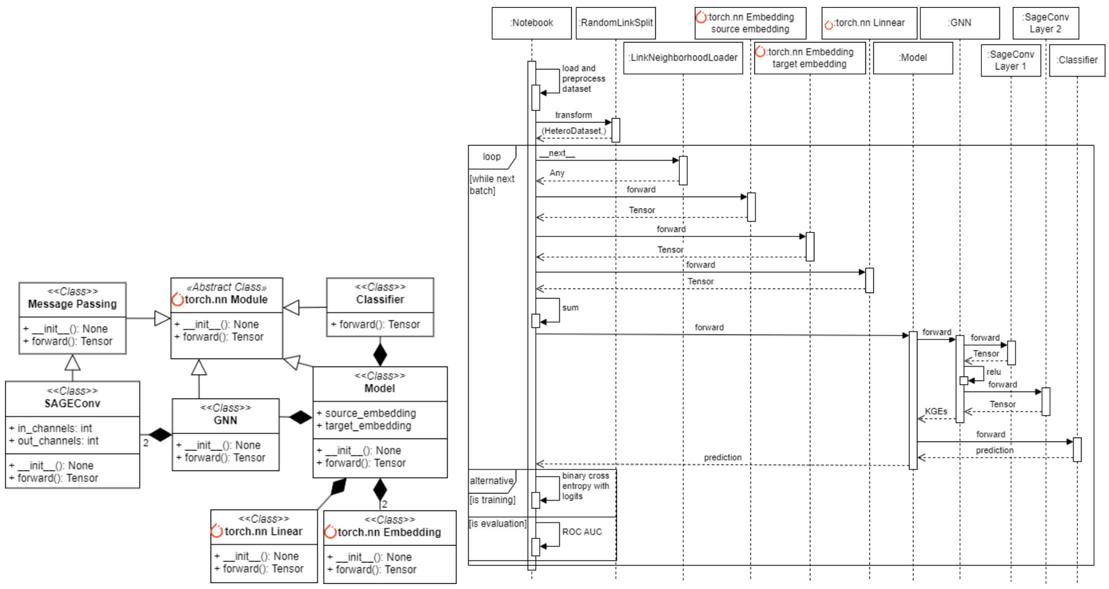
dataset. During the main training loop ***train()*** the trainer generates ***DataLoaders*** for training and evaluation datasets using the *DataCollatorForLanguageModeling*. A *DataLoader* provides an iterable over the batched datasets. With ***create_optimizer_and_schedular***, the trainer also initializes the optimizer with the parameters of the PretrainedModel.
Tokenizers are model specific and produce the numerical encodings of input texts with their attention mask. Tokenizers can use padding and truncation during the transformation process to even out the length of every encoding. Tokenizers can be adjusted during the transformation process to produce additional properties, like segments (*token types ids*). Tokenizer lives independently of the entire training process. Each tokenizer inherits from ***PreTrainedTokenizerBase***, which again implements basic functions, like loading pretrained[7] tokenizers. ***PreTrainedTokenizer*** implements the main tokenize processing pipeline, that is shared across all tokenizers, while ***BertTokenizer*** inherits from *PreTrainedTokenizer* and initializes the tokenizer with the BERT (Devlin et al. [2019]) special tokens *[CLS]*, *[MASK]*, *[SEP]*, *[PAD]* and *[UNK]*. *BertTokenizer* is build on top of ***WordPieceTokenizer*** (Wu et al. [2016]).

2.5.5 PyTorch Geometrics

PyTorch Geometrics (PyG) (Fey and Lenssen [2019]) is a geometric deep learning extension for PyTorch. It leverages a simple message passing API for efficient GNN and GCN implementations like GraphSage (Hamilton et al. [2017]). ***SAGEConv*** is an operator that implements a GraphSage layer (Hamilton et al. [2017]). ***HeteroData*** is a data object describing heterogeneous graphs. This data object behaves like a nested dictionary with basic torch functionalities. The object can hold node-level or link level attributes. Datasets like HeteroData can be edge-level split into training, evaluation and test with ***transforms.RandomLinkSplit***. Negative edges (node pairs without edge) can be added automatically as well by this transformation method. ***LinkNeighborLoader*** is a class that allows sampling linked sub-graphs of large graphs, that are not feasible to train with otherwise and also generate negative on the fly.

PyG Tutorials were published under the MIT licence on the PyG website[8]. The tutorial for *link prediction*[9] introduces the reader to GNNs with PyG.

[7]Hugging Face chose the term "pretrained" in this context, probably because the pretrained models were trained on given tokenizer configuration and dictionary.
[8]Link to the Tutorials: https://pytorch-geometric.readthedocs.io/en/latest/get_started/colabs.html
[9]Link to the tutorial for link prediction:
https://colab.research.google.com/drive/1xpzn1Nvai1ygd_P5Yambc_oe4VBPK_ZT?usp=sharing



(a) Class diagram of the link prediction model and it's GNN (GraphSage) components (b) Sequence diagram of the link prediction training pipeline

Figure 2.12: The PyG link prediction tutorial as in *https://pytorch-geometric.readthedocs.io/en/latest/get_started/colabs.html*, by PyG Team © Copyright 2025, released under MIT license

Figure 2.12 illustrates the PyG link prediction tutorial. The authors first load and preprocess the MovieLens dataset (Harper and Konstan [2015]). For that they assign users and movies each IDs in range. The genres of movies are also one-hot encoded. Ratings and tags were dismissed. Then they define the *GNN* model as depicted in figure 2.12a. A ***SageConv*** implements GraphSage layers (Hamilton et al. [2017]), which ultimately inherits from *torch.nn.Module*. The ***GNN*** class also inherits from *torch.nn.Module* and holds two references of *SageConv* layers. The ***Classifier*** also inherits from *torch.nn.Module*, but does not contain any trainable parameters. Instead, it calculates the dot product between two given tensors. The ***Model*** class wraps *GNN* and *Classifier* into a fully functional GraphSage model. For that, it uses two *torch.nn.Embedding* modules for source and target initial embeddings, as well as a *torch.nn.linear* layer for the one-hot encoded target features.

Figure 2.12b illustrates the training pipeline in a sequence diagram. After preprocessing the dataset, RandomLinkSplit transform the dataset into three splits *train, test, validation* and adds false edges to the *test* and *validation* split. For each batch in *LinkNeigborhoodLoader* a batch of neighborhoods is fetched and false edges are generated for the test split on the fly. Then the source and target IDs are embedded via *torch.nn.Embedding* and the target features are embedded via *torch.nn.linear*. The target ID and feature embeddings are them summed up. This results in the source and target initial vector representations. The model receives a batch of neighborhood initial vector representations and passes them into the first *SageConv* layer, followed by ReLU followed by the second *SageConv* layer. This results in vector representations (KGEs) of source and target nodes as discussed in GRL. These KGEs are then passed to the classifier and the dot product is applied on them. The resulting scalar is then passed with the ground truth, 0 for no edge in between source and target node and 1 for edge between source and target node into a binary cross entropy with logits loss function that fits the models prediction the the ground truth. During evaluation this process is replaced with a *ROC AUC* metric.

2.5.6 Scikit-Learn

Scikit-Learn (Pedregosa et al. [2011]) is based on *Numpy* and provides an easy-to-use interface for many state-of-the-art machine learning implementations. *Scikit* implements PCA, multiple metrics for model performances, like accuracy, f1 score or ROC AUC and can be used to compute and visualize the confusion matrix.

2.5.7 Matplotlib

Matplotlib (Hunter [2007]) is a plotting library based on Numpy and MATLAB. *Matplotlib* supports 2 and 3-dimensional plots, such as line or scatter plot.

2.5.8 NetworkX

NetworkX (Hagberg et al. [2008]) provides a flexible data structure for many graph types, such as (un-)directed graphs with loops and complex node data types. Graphs in *NetworkX* can be visualized by an interface to the *Matplotlib* library with simple node positioning.

2.6 Research Gaps

LLMs show great potential in numerous downstream tasks, like question answering, common-sense reasoning, mathematic science and coding OpenAI [2023], Gemma Team and Google DeepMind [2024], Llama Team and AI @ Meta [2024], DeepSeek-AI [2025]. Widely used pretrained-transformer still show many disadvantages when dealing with graphs (Chen et al. [2024b]). That is why there are multiple strategies to mix GNN architectures with transformer architectures (Jin et al. [2024], Chen et al. [2024b]). While there are multiple strategies mixing

GNN with transformer architecture suggested, the evaluation tends to be based purely on performance (Mohammadkhani et al. [2023]), despite the increasing demands on XAI (Speith [2022], Singh et al. [2024], Zhao et al. [2024], Volkov and Averkin [2024]). From our knowledge there is only one scientific publication using XAI methods to understand the integration of GRL in LLMs in detail (Li et al. [2024]).

In this thesis we are analyzing the semantics learned by a BERT encoder model (Turc et al. [2019]) that is faced with KGEs produced by a GraphSage model (Hamilton et al. [2017]) and introduced via soft prompts as in *GraphPrompter* (Liu et al. [2024b]) with XAI methods. The insights we are producing help us understand what semantic features the BERT model learns to interpret from the KGEs produced by the GraphSage model. We will also be able to check for any unwanted behaviors like biases, that stem from the fusion of these different views, which may stay unrecognized if the evaluation was purely based on performance.

3 Tool Suite

We are proposing a tool suite, that fuses multiple concepts and python libraries. There is a technical standard we try to uphold, which stems from the availability of open source projects and their embedding into well-established python libraries, such as PyTorch (Paszke et al. [2019]) in general, *Hugging Face Transformers and Datasets* (Wolf et al. [2020], Lhoest et al. [2021]) for NLP and *PyG* (Fey and Lenssen [2019]) for GRL. We hope, by providing source code, that leverages said libraries, results into a higher accessibility. We also came to the conclusion, that despite theoretical plans, the actual implementations may come with unwanted behaviors, due to the complexity of said libraries and their tuning potential. We belief to make this tool assessable to a broader audience, projects, such as this, need to be fire-tested in the actual runtime environment. We made sure to include all python libraries that were used in this project in the previous chapter. We also included detailed implementation details of classes and other projects in case we adept or change them in this tool suite.

The goal of this tool suite is to provide a pipeline, that is producing KGEs with a GraphSage (Hamilton et al. [2017]) PyG model, pass those KGEs to a $BERTTiny_{128\_2}$ model Devlin et al. [2019], Turc et al. [2019] via soft-prompts such as in GraphPrompter (Liu et al. [2024b]) and then use XAI methods to analyze the transformer's behavior. We call our tool suite **GraphPrompter for Hugging Face (GraphPrompterHF)**. GraphPrompterHF has some conceptual changes to the original GraphPrompter architecture tho. Figure 3.1 illustrates the GraphPrompterHF changes compared to original GraphPrompter (Liu et al. [2024b]). Most changes stem from our XAI methods, that we need to apply, in order to understand the transformer's internal semantic behavior. We mix several concepts of XAI methods in order to achieve comprehensible results with high performance. The basis for all our analyses is classifier-based probing. GraphPrompterHF will be trained on the GRL downstream-task *link prediction*, which allows us to increase the need on processing graph related properties by the transformer. Because we want to leverage XAI methods, such as SHAP (Lundberg and Lee [2017]), we base the fine-tuning of GraphPrompterHF on multiple conceptual text segments, which later allows us to use *concept-based explanations* as in Mohammadkhani et al. [2023] and Kokalj et al. [2021].

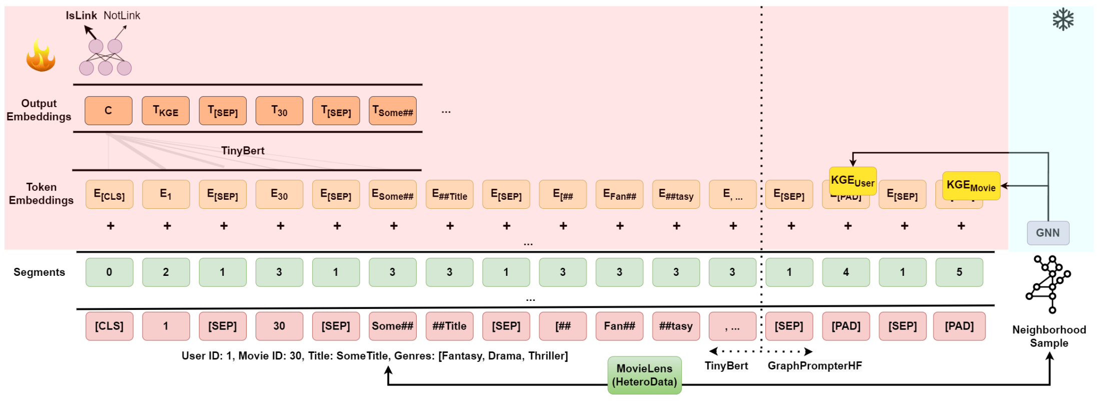


Figure 3.1: GraphPrompterHF architecture on the task of link prediction adapted from *Can we Soft Prompt LLMs for Graph Learning Tasks?*, by Lio et al., 2024 and *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*, by Devlin et al., 2019

BERT (Devlin et al. [2019]) introduced *segment* embeddings (*token_type_embeddings* in Hugging Face) to the transformer architecture, to help the transformer separate between the sentences in NSP. We increase the number of segments from 2 to 6, so that we can leverage the segments in XAI implementations and help the transformer distinguish between source and target node features, KGEs and texts and special tokens, like *[CLS]* and *[SEP]*.
For SHAP (Lundberg and Lee [2017]) we remove tokens from the input sequence by masking them with the mask token *[MASK]*, as already suggested by TransSHAP (Kokalj et al. [2021]). We do not mask singular tokens but we group tokens by their segment (*token_type_id*), reducing the complexity of candidates and increasing the interpretability similar to the approach of Kokalj et al. [2021].
For the GNN, we made sure, that the output dimensionality of the GNN fit the input embedding dimensionality of the transformer model, which allowed us to remove the projection layer between GNN and transformer. The downstream-task was chosen in a way, so that the GNN performs much better then the transformer. In this setup, we wanted to analyze if and how the transformer is able to leverage the much stronger GNN performance by understanding the features of KGEs. That is why we chose to keep the GNN frozen and instead trained the transformer model during fine-tuning. We also added another 2 phases of fine-tuning, one before, in which the transformer and GNN models are trained separately on the dataset and one after the soft prompt fine-tuning, in which the entire GraphPrompterHF was trained end-to-end.
We based the GNN link prediction implementation on the *PyG link prediction tutorial*[1] and

[1]https://pytorch-geometric.readthedocs.io/en/latest/get_started/colabs.html

adjusted some of its hyper-parameters to our use-case and used the main (larger) version of the MovieLens dataset (Harper and Konstan [2015]).

3.0.1 GraphPrompterHF

GraphPrompter (Liu et al. [2024b]) showed that leveraging soft prompts for KGEs in LLMs can be a serious approach. While the approach shows promises, the actual implementation[2] is more of a proof of concept. The implementations performance can be enhanced by leveraging more tools of PyTorch Tensor. For example, in the current implementation of the *forward pass*, every word embedding in a batch is produced sequentially. One improvement could be to produce the word embeddings for the entire batch in parallel. Because KGEs are no longer concatenated at the beginning of the sequence, but at the end and because the sequences have difference lengths, we are forced to use different mechanics how KGEs and LLM input embeddings are concatenated. For that we can leverage the segments (*token_type_ids*) and padding tokens (*[PAD]*).
We construct the input sequence in a way, that the positions of the KGEs are first taken by padding tokens. Then during the forward pass, we generate the actual KGEs with the GNN and replace the padding tokens with the KGEs by using the segments as a mask. Given the input features "*User ID: 1, Movie ID: 30, Title: SomeTitle, Genres: [Fantasy, Drama, Thriller]*" results in the input prompt: "*[CLS]1[SEP]30[SEP]SomeTitle[SEP][Fantasy, Drama, Thriller][SEP][PAD][SEP][PAD][SEP]*" and a segment sequence (*token_type_ids*) of: *[0,2,1,3,1,3,3,1,3,3,3,3,3,3,1,4,1,5,1].*

$$\vec{\mathbf{T}}_i = \begin{cases} 1, & \text{if } \vec{\mathbf{S}} = \mathbf{S}_i \\ 0, & \text{otherwise} \end{cases} \tag{3.1}$$

$$\vec{\mathbf{M}}_i = \vec{\mathbf{T}}_i \vec{\mathbf{1}}^{\mathsf{T}}_{\mathbf{d}_H} \tag{3.2}$$

$$\overrightarrow{\mathbf{KGE}}_i = \overrightarrow{\mathbf{kge}}_i \vec{\mathbf{1}}^{\mathsf{T}}_{d_S \times d_H} \tag{3.3}$$

$$\vec{\mathbf{W}} = \left(\vec{\mathbf{W}} \odot (\vec{\mathbf{1}} - \vec{\mathbf{M}}_s) + \overrightarrow{\mathbf{KGE}}_s \odot \vec{\mathbf{M}}_s\right) \odot (\vec{\mathbf{1}} - \vec{\mathbf{M}}_t) + \overrightarrow{\mathbf{KGE}}_t \odot \vec{\mathbf{M}}_t \tag{3.4}$$

Where
$i \in [s, t]$ is an index for either source or target node vectors,
$d_H \in \mathbb{R}_{>0}$ is the hidden size of the model ($d_H = 128$ for $BERTTiny_{128}$ (Turc et al. [2019])),
$d_S \in \mathbb{R}_{>0}$ is the sequence length dimension of the model ($d_S = 512$ for all BERT models (Turc

[2]Link to the original GraphPrompter implementation: https://github.com/franciscoliu/graphprompter

et al. [2019])),
$\vec{\mathbf{S}} \in \mathbb{R}^{d_B \times d_S}$ is the segment vector (*token_type_ids*) of the batch,
$\vec{\mathbf{S}}_i \in \mathbb{R}$ are the segments of source and target (here 4 and 5),
$\vec{\mathbf{1}}_{d_H}$ is the unit vector of dimension d_H,
$\vec{\mathbf{1}}_{d_S \times d_H}$ is the unit tensor of dimension $d_S \times d_H$,
$\overrightarrow{\mathbf{kge}}_i \in \mathbb{R}^{d_B}$ are the KGEs of source and target nodes over the entire batch,
$\vec{\mathbf{1}}$ is the unit tensor of dimension $(d_B \times d_S \times d_H)$ and
$\vec{\mathbf{W}} \in \mathbb{R}^{d_B \times d_S \times d_H}$ are the word embeddings.
We filter the segments (*token_type_ids*) for source and target KGE positions in equation 3.1. Then we scale (by repeating) both segment mask $\vec{\mathbf{T}}$ and KGEs over the batch $\overrightarrow{\mathbf{kge}}$ to dimension $(d_B \times d_S \times d_H)$ in equations 3.2 and 3.3. Last equation 3.4 we use the segment masks and their inverse to replace some word embedding positions with KGEs, exactly where the segments are equally the segment id for source and target KGE positions.
This implementation is differentiable regarding backpropagation and can be implemented completely in PyTorch operations, like addition, multiplication and matching, as well as reshaping with *torch.tensor.unsqueeze* and *torch.tensor.repeat*.

3.0.2 Group By Segments

Grouping token positions by (semantic) features such as with *concept-based explanations* (Zhao et al. [2024]) for attention maps (Mohammadkhani et al. [2023]) and TransSHAP (Kokalj et al. [2021]) allows us to reduce the complexity and memory requirements of XAI methods, increase their performance and provide the reader with a (semantically) summarized insight. Our approach to expand the different segments for this structured dataset does not only allow us to insert KGEs in the input embeddings as in equation 3.4, it also allows us to group internal vector representations (*hidden_states*) by token positions as well as replacing entire segment groups (for SHAP (Lundberg and Lee [2017]) as already done with other mechanics in TransSHAP (Kokalj et al. [2021]).

Internal Vector Representations (Hidden States) by Segments

Grouping hidden states by segments reduces its memory requirement. Instead of storing the hidden states over all positions we store hidden states ones for each segment. Grouping hidden states by segments allows us to make *global explanations* about vector representations of segment groups. We group hidden states by segments by averaging over all hidden states with the same segment. For $SEG = [1,2,3,4,5,6]$ and every segment $i \in SEG$ we calculate the segment masks $\vec{T}_i$ and $\vec{M}_i$ as in equations 3.1 and 3.2.

$$\vec{\mathbf{H}}_i = \frac{\sum_{d_S}\left(\vec{\mathbf{H}} \odot (\vec{\mathbf{M}}_i \vec{\mathbf{1}}^{\intercal}_{d_L})\right)}{\sum_{d_S}(\vec{\mathbf{T}}_i \vec{\mathbf{1}}^{\intercal}_{\mathbf{d}_L})} \tag{3.5}$$

$$\vec{\mathbf{H}}_{mean}(b,i,l,h) = \vec{\mathbf{H}}_i(b,l,h) \tag{3.6}$$

Then we calculate the average of each segment hidden state $\vec{\mathbf{H}}_i \in \mathbb{R}^{d_B \times d_H \times d_L}$ of each attention layer $d_L \in \mathbb{R}_{>0}$ ($d_L = 3$ for $BERTTiny_{128\_2}$ (Turc et al. [2019])) by summing all hidden states $\vec{\mathbf{H}} \in \mathbb{R}^{d_B \times d_S \times d_H \times d_L}$ over the sequence length d_S masked by the segment masks $\vec{\mathbf{M}}_i$ and divide the sum element wise by the sum of the segment mask $\vec{\mathbf{T}}_i$ over dimension d_S of as shown in equation 3.5. Simply speaking, we sum the hidden states of all positions in the segment and divide them by the amount of positions in the segment resulting in one hidden state for each segment for each data point in the batch for each attention layer.
Then we stack the resulting average hidden states over the first dimension as shown in equation 3.6, where $b \in [0, d_B)$, $l \in [0, d_L)$ and $h \in [0, d_H)$, resulting in a vector $\vec{\mathbf{H}}_{mean} \in \mathbb{R}^{d_B \times |SEG| \times d_H \times d_L}$. We use PCA (Pearson [1901], Hotelling [1933]) on any chosen layer l and segment i over the entire batch of hidden states to reduce the hidden states to 2-dimensions with their 2 most important components and plot them on scatter plots as illustrated in figure 2.9. We also manipulate the color and markers of these points based on ground truth features, like data point's label or **movie popularity** (in degree of node).

Attention Maps by Segments

As Mohammadkhani et al. [2023] already suggested, grouping attention maps by concepts for structured data, like source code increases its interpretability and allows to make global statements based on local observations. The attention map in its original form is the attention score, which is based on the dot product between key and query of the token positions. That is why in every layer we are looking at d_S^2 attention scores. We will reduce the complexity to $|U(\vec{\mathbf{S}})|^2$, meaning we are looking for the key query pairs of segments instead of token positions.

$$\vec{\mathbf{A}} = \frac{\sum_{d_{AH}} \vec{\mathbf{A}}_{AH}}{d_{AH}} \tag{3.7}$$

$$\vec{\mathbf{A}}_i^{source} = \frac{\sum_{d_S^{source}}\left(\vec{\mathbf{A}} \odot (\vec{\mathbf{M}}_i \vec{\mathbf{1}}^{\intercal}_{d_S \times (d_L-1)})\right)}{\sum_{d_S^{source}}(\vec{\mathbf{T}}_i \vec{\mathbf{1}}^{\intercal}_{d_S \times (d_L-1)})} \tag{3.8}$$

$$\vec{\mathbf{A}}_{mean}^{source}(b,i,s,l) = \vec{\mathbf{A}}_i^{source}(b,s,l) \tag{3.9}$$

$$\vec{\mathbf{A}}_j^{target} = \frac{\sum_{d_S^{target}} \left(\vec{\mathbf{A}}_{mean}^{source} \odot (\vec{\mathbf{M}}_j \vec{\mathbf{1}}^{\top}_{(d_L-1)\times|SEG|})\right)}{\sum_{d_S^{target}} (\vec{\mathbf{T}}_j \vec{\mathbf{1}}^{\top}_{(d_L-1)\times|SEG|})} \tag{3.10}$$

$$\vec{\mathbf{A}}_{mean}(b,i,j,l) = \vec{\mathbf{A}}_j^{target}(b,i,l) \tag{3.11}$$

Computing the average attention masks over segments is very similar to the hidden states process, but this time we are facing attention maps for every attention head $\vec{\mathbf{A}}_{AH} \in \mathbb{R}^{d_B \times d_{AH} \times d_S^{source} \times d_S^{target} \times (d_L-1)}$, which we compute the average of at the beginning in equation 3.7 with d_{AH} being the amount of attention heads in the transformer. With this, we compute $\vec{\mathbf{A}} \in \mathbb{R}^{d_B \times d_S^{source} \times d_S^{target} \times (d_L-1)}$, where each attention score is the score between key-query token position pairs and the the last layer is excluded, because there is no attention score for that layer. First we compute the average of all positions in the same group i resulting in $\vec{\mathbf{A}}_i^{source} \in \mathbb{R}^{d_B \times d_S^{target} \times (d_L-1)}$ as seen in equation 3.8. Then we stack those vectors over dimension 1 resulting in $\vec{\mathbf{A}}_{mean}^{source} \in \mathbb{R}^{d_B \times |SEG| \times d_S^{target} \times (d_L-1)}$ as seen in equation 3.9. We repeat this procedure on the already averaged attention map, over the same segments $j \in SEG$ and the sequence length d_S^{target} resulting in attention scores from each segment i to each other segment j with $\vec{\mathbf{A}}_j^{target} \in \mathbb{R}^{d_B \times |SEG| \times \times (d_L-1)}$ as seen in equation 3.10. Then in equation 3.11, we stack those vectors over dimension 2, resulting in $\vec{\mathbf{A}}_{mean} \in \mathbb{R}^{d_B \times |SEG| \times |SEG| \times (d_L-1)}$.

The attention map is then plotted in a bipartite graph as suggested by Zhao et al. [2024] and Vig [2019]. All attention scores are normalized and then scaled by some arbitrary static value to make the lines more human readable.

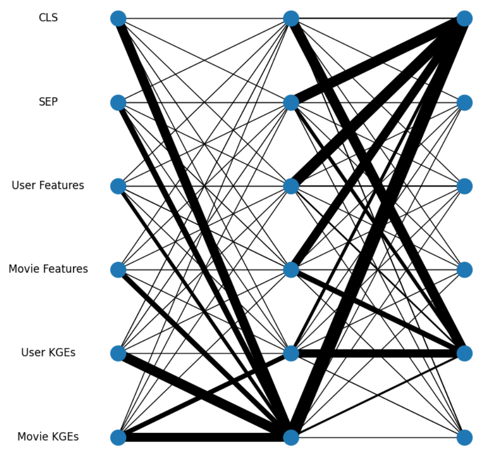


Figure 3.2: Illustration of an attention map.

Figure 3.2 illustrates the attention map of a sampled GraphPrompterHF forward pass. As we can see, there are the 6 segments of any datapoint and we can see the two layer of multi-head self-attention blocks that result in the output of the classifier token (*[CLS]*). As we can see, sometimes segments are paying attention to themselves, but for the most part this is not necessary, because of the residual layer (He et al. [2016]) around each attention-block. In other words, a low attention score on a key-query pair where the key equals the query does not mean, that this information is lost, making the attention map susceptible to misinterpretation.

SHAP Mask by Segment Group

For that, we replace the segment mask 3.1 with a segment group mask.

$$U(\vec{\mathbf{S}}) = \{S_i | S_i \in \vec{\mathbf{S}}\} \tag{3.12}$$

$$P(C) = \{C | C \subseteq U(S)\} \tag{3.13}$$

$$\vec{T}_i = \begin{cases} 1, & \text{if } \vec{\mathbf{S}} \in P(C) \\ 0, & \text{otherwise} \end{cases} \tag{3.14}$$

Equation 3.1 is replaced with equations 3.12, 3.13 and 3.14. This time we calculate the set of all possible unique segments in equation 3.12, then we calculate the set of all possible combinations of segments in equation 3.13 and then calculate the mask of token positions in a specific segment combination with equation 3.14. Equation 3.12 and 3.13 can be computed only ones before the training, because in this structured dataset every possible segment is predetermined.

$$\vec{I} = \vec{I} \odot (\vec{1}_{d_B \times d_S} - \vec{T}_i) + (\vec{I}_{[MASK]} * \vec{1}_{d_B \times d_S}) \odot \vec{T}_i \tag{3.15}$$

In equation 3.15, where $\vec{I} \in \mathbb{N}^{d_B \times d_S}$ is the numerical representation of the input sequence (*input_ids*) and $\vec{I}_{[MASK]} \in \mathbb{N}$ is the input id of the mask token (*[MASK]*), we replace some of the input ids with mask tokens and this way remove features with given segments (*concepts*) from the prediction. Unlike inserting KGEs into input embeddings, we do not need to scale the segment masks $\vec{T}_i$, because we do not operate on embedding dimension d_H.

We compute the SHAP-values (Lundberg and Lee [2017]) on the validation dataset split after the training on unseen data to extract get the importance of all segments. With every segment-grouped masked, we do not only compute the SHAP-Value, but also store the averaged attention maps and hidden states, so we can analyze the internal behavior given some segments are masked.

3.0.3 Three Training Stages of GraphPrompterHF

We are analyzing three stages of the same transformer model. First, the *Vanilla* model which consist of only $BERTTiny_{128\_2}$ (Turc et al. [2019]) and uses the natural language portion of the dataset is trained on the task of link prediction. For that we reduce the amount of possible segments (SEG) from 6 to 4, removing the segments concerned with KGEs. In parallel, we train the GNN (GraphSage (Hamilton et al. [2017])) as already implemented in the PyG link prediction tutorial. In the second stage, we are continue training *Vanilla* and GNN model in their composition as frozen GraphPrompterHF model, where the GNN's parameters are kept frozen. On top of the frozen GraphPrompterHF model, we are training both, GNN and transformer, end-to-end. Last, we analyze all model stages (Vanilla, GraphPrompterHF frozen and GraphPrompterHF end-to-end) with XAI.

3.1 Architecture

The components in the tool suite are separated by concerns and enables clear interchangeability of datasets and models. The clear boundaries between components also allow the production of intermediate pipeline step artifacts. The pipeline can be stopped and repeated at any step.

3.1.1 Pipeline

The tool suite's pipeline is as follows:

1. Loading and transforming dataset into standardized format.
2. train GraphSage (GNN) and Vanilla model
3. train GraphPrompterHF frozen
4. train GraphPrompterHF end-to-end
5. produce XAI artifacts
6. evaluate XAI artifacts

The first part of the pipeline is the loading and transformation of the dataset in standardized format. Then the GNN model and Vanilla Model are trained. GraphPrompterHF frozen is then trained on GNN and Vanilla model. Then GraphPrompterHF end-to-end is trained on GraphPrompterHF frozen. Last, the XAI artifacts are produced by the LLMs and evaluated. Most of the components concerns can be directly derived from these pipeline steps. Thus, we are further discussing each pipeline step individually, so that the inputs and outputs are better defined.

Dataset Format

Every dataset needs to end up in a specific format. We are using Pandas DataFrames (Mc Kinney [2010]) for handling the datasets ids and features. Each dataset has to contain *source_id* and *target_id*, which are both consecutive integer ranges starting with zero. Each datapoint is therefor described by the *source_id* node and *target_id* node. Every feature of these datapoints are also separated into *source*-features and *target*-features. In addition, we separate features into features for NLP and for structural graph processing. For either model, the column name of a feature starts with the target model, *llm_* or *gnn_* , either *source_feature_* or *target_feature_* followed the actual feature name. Let's say we have a feature named "population", that is used in NLP and concerns a source node, then we end up with the column name: $llm\_source\_feature\_population$. The structural (GNN) features can only take numerical values, in our case categorical one-hot-encoding of genres.

source_id	target_id	[llm | gnn]_[source | target]_feature_<name>
0	10	<feature>
0	22	<feature>
1	3	<feature>

Table 3.1: Dataset standardized format

The resulting pandas DataFrames should look like in table 3.1. The third column represents all possible combinations of LLM, GNN, source and target features. Ones this format is met, we can generate the HeteroData object of PyG (Fey and Lenssen [2019]), split the dataset into *train*, *test* and *validation*, add non existing edges (combination of source and target nodes that do not share an edge between each other) to the splits *test* and *validation*.

3.1.2 Components

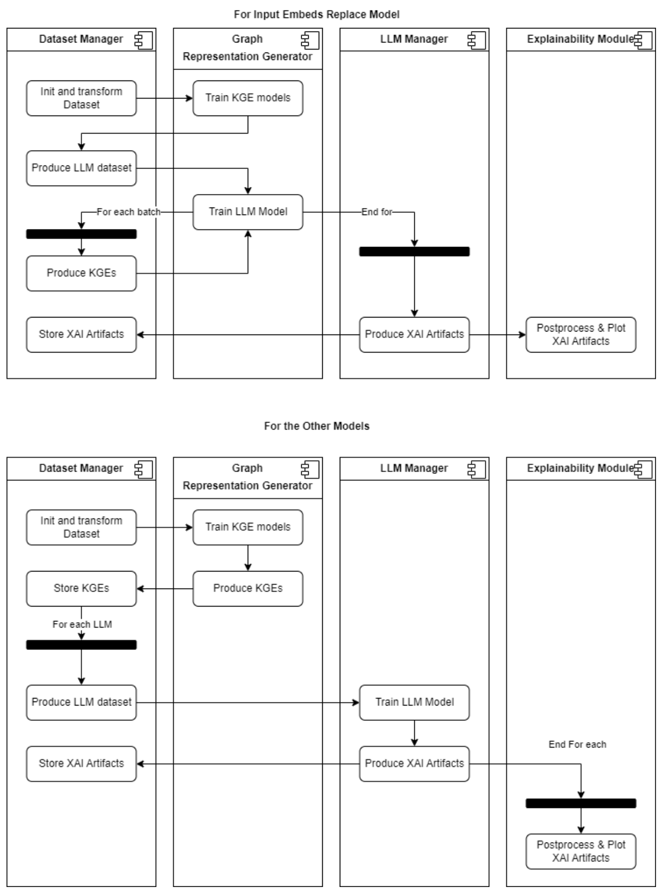


Figure 3.3: Components in the pipeline

The pipeline and requirements outline roughly the components for this tool suite's architecture. We want to make the components interchangeable, have clear boundaries and intermediate artifacts between pipeline steps as depicted in figure 3.3. First comes the **Dataset Manager** a component that manages the datasets transformation into a normalized formats, and the storage and retrieval of all dataset related artifacts. Next, the **Graph Representation Generator** that is fully responsible for the GNN models and for providing an interface for generating KGEs. The **LLM Manager** is responsible for the LLMs, like training them. Lastly the **Explainability Module** is responsible for post processing and plotting all XAI artifacts generated by the **LLM Manager**.

Dataset Manager

The **Dataset Manager** manages the format and storage processes of the tool suite. Some of these steps are dataset dependent. For example, how to load the original dataset, and its structure is highly dataset dependent. That is why the **DatasetManager** implements one abstract class with all processes, that are dataset independent and which work on the normalized format. This abstract class expects three pandas DataFrames in a particular format.
The **source-** and **target-DataFrames** contain the consecutive source- or target-ids and features of each node in the KGE, with the same signature as described in table 3.1. Then there is an **edge-DataFrame** that contains all source- and target-id pairs.
Once these three DataFrames are generated, they can be passed to the constructor of the abstract class, which implements all dataset independent functions. In this constructor, the three DataFrames are merged to produce the DataFrame as in table 3.1. This merged DataFrame will be the one used by the LLMs, while the source-, target- and edge DataFrame will be used by the GNN dataset. These GNN datasets (HeteroData objects) are now generated and split.

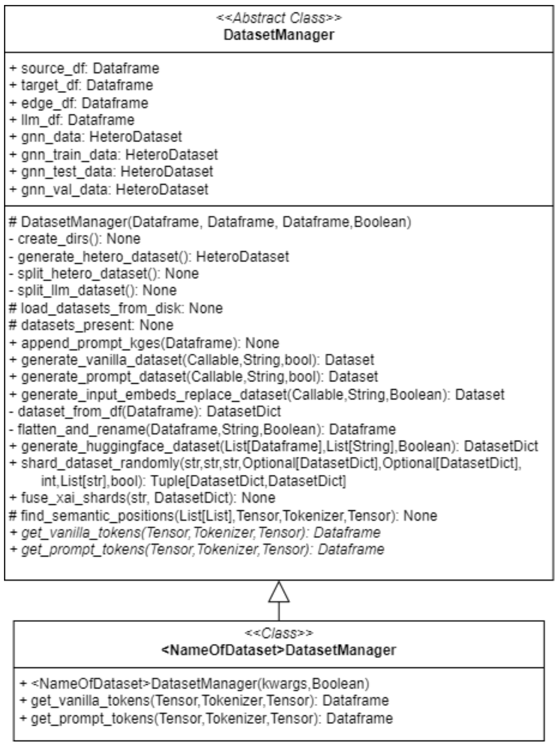

<<Abstract Class>>
DatasetManager
+ source_df: Dataframe
+ target_df: Dataframe
+ edge_df: Dataframe
+ llm_df: Dataframe
+ gnn_data: HeteroDataset
+ gnn_train_data: HeteroDataset
+ gnn_test_data: HeteroDataset
+ gnn_val_data: HeteroDataset
# DatasetManager(Dataframe, Dataframe, Dataframe,Boolean)
- create_dirs(): None
- generate_hetero_dataset(): HeteroDataset
- split_hetero_dataset(): None
- split_llm_dataset(): None
# load_datasets_from_disk: None
# datasets_present: None
+ append_prompt_kges(Dataframe): None
+ generate_vanilla_dataset(Callable,String,bool): Dataset
+ generate_prompt_dataset(Callable,String,bool): Dataset
+ generate_input_embeds_replace_dataset(Callable,String,Boolean): Dataset
- dataset_from_df(Dataframe): DatasetDict
- flatten_and_rename(Dataframe,String,Boolean): Dataframe
+ generate_huggingface_dataset(List[Dataframe],List[String],Boolean): DatasetDict
+ shard_dataset_randomly(str,str,str,Optional[DatasetDict],Optional[DatasetDict], int,List[str],bool): Tuple[DatasetDict,DatasetDict]
+ fuse_xai_shards(str, DatasetDict): None
# find_semantic_positions(List[List],Tensor,Tokenizer,Tensor): None
+ get_vanilla_tokens(Tensor,Tokenizer,Tensor): Dataframe
+ get_prompt_tokens(Tensor,Tokenizer,Tensor): Dataframe
<<Class>>
<NameOfDataset>DatasetManager
+ <NameOfDataset>DatasetManager(kwargs,Boolean)
+ get_vanilla_tokens(Tensor,Tokenizer,Tensor): Dataframe
+ get_prompt_tokens(Tensor,Tokenizer,Tensor): Dataframe

Figure 3.4: DatasetManager class diagram

The class diagram in figure 3.4 shows the abstract class **DatasetManagerBase** and an interface **DatasetManagerInterface**. The interface defines what an implementing class of a specific dataset has to implement. The constructor of each class has to accept at least a flag *[TRUE/-FALSE]*, for skipping the loading and pre-processing of the dataset, if the expected outputs can already be found on the disk. If *TRUE*, the recomputation of every step is forced. If *FALSE* (default), the recomputation is skipped, if this step was already performed before and then, all datasets are loaded from disk.
The constructor of any implementing class also have to call the constructor of the abstract class. Because this constructor expects the three *pandas DataFrames: source_df, target_df and edges_df* as described before, the constructor of the implementing class is forced to pass them in the expected format.

Attributes

DatasetManagerBase has multiple attributes, all representing the dataset from multiple views and types. We have already discussed the attributes *source_df*, *target_df*, *edge_df* and *llm_df*. The other HeteroData objects are the entire dataset for the GNN model: *gnn_data* and the three splits of that dataset: *gnn_train_data*, *gnn_test_data* and *gnn_val_data*.

DatasetManagerBase Constructor

As discussed the constructor receives the three DataFrames and the flag, to force a recompute. The private methods *create_dirs*, *generate_hetero_dataset*, *split_hetero_dataset* and *split_llm_dataset* are called in the constructor and run the most preprocessing steps on the dataset. For splitting the HeteroData, we are using the PyG *RandomLinkSplit*[3], a disjunct edge-level random split that add non-existing edges to the test and validation split.

File Structure

After the DatasetManager is generated, the file structure in the data-root is as depicted in figure 3.5. There is a folder for all GNN related data, like the HeteroData(-splits) and a folder for all LLM related data, like the DataFrames and and the folders where all LLM training processes and model checkpoints are saved. There can be also some files and folders, that an implementing DatasetManager are loading, unwrapping for pre-processing purposes. The protected methods *load_dataset_from_disk* and *dataset_present* rely on the given file structure to check and load given dataset objects.

[3]URL to RandomLinkSplit:
https://pytorch-geometric.readthedocs.io/en/2.4.0/generated/torch_geometric.transforms.RandomLinkSplit.html

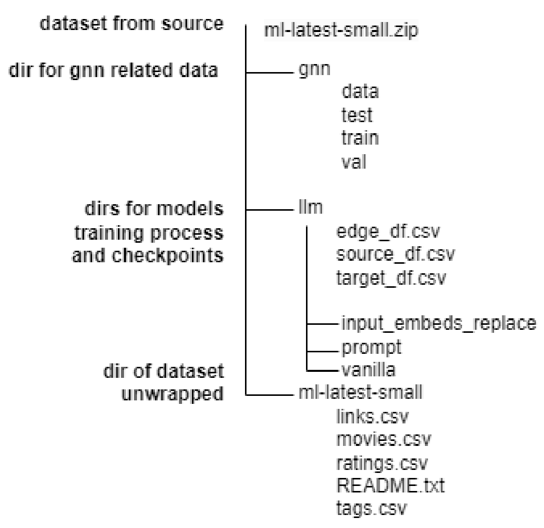


Figure 3.5: File structure after Initialization

Hugging Face Datasets (Lhoest et al. [2021]) are produced by the two methods *generate_vanilla_dataset* and *generate_graph_prompter_hf_dataset* for the respective LLM model. Each method receives a tokenize callback function of the LLM component. We explain this function in detail later in the LLM manager component. The second parameter is the separation token used by the models tokenizer. This separation token (default *[SEP]*) is inserted between each id and feature during input prompt generation. And again, the last parameter is a flag, whether the dataset generation is forced to be recomputed.
The methods *generate_huggingface_dataset* is used at the end of the pipeline, when all XAI artifacts have been generated. The method receives a list of DataFrames with the XAI artifacts a,d a list of prefixes. The DataFrames' columns can be assigned to their source after merge and a flag, if the the features and ids in clear format are to be included or not. Again, we are explaining the XAI artifact DataFrames later in the LLM component.
The method *flatten_and_rename* is used multiple times by *generate_huggingface_dataset* to add the prefixes to the respective columns and to flatten any arrays. For each array flattend, a shape array is added with the original shape of the arrays, so they can be transformed back into their original state. This is necessary, because Hugging Face does not allow nested arrays in their data fields.

Pre- and post-processing steps of the evaluation outputs of the models get handled by *shard_dataset_randomly* and *fuse_xai_shards*. The first method slices given *Vanilla* and *Graph-*

PrompterHF datasets into the given shard size and saving the resulting datasets on disk. We use this method so we do not have to evaluate the entire dataset.
The other method collects all evaluation datasets of the *SHAP-value* calculation (one dataset for each model and segment group) and fuses them into a single dataset for each model and segment. This step reduces the memory requirement of the evaluation datasets, which would not fit into memory otherwise.

Graph Representation Generator

The component **Graph Representation Generator** is responsible for managing the GNN models and producing KGEs. The class **GraphRepresentationGenerator** is responsible for managing one GNN model. It provides an interface for initializing/loading a model, train it and produce KGEs for one or multiple passed node ids. Each GraphRepresentationGenerator holds one instance of **LinkPredictionModel** class. The LinkPredictionModel class is inheriting from PyTorch's *nn.Module*. The LinkPredictionModel composites one of each instance of the **GNN** and **Classifier** classes. Both, GNN and Classifier classes inherit from PyTorch's *nn.Module* class as well.

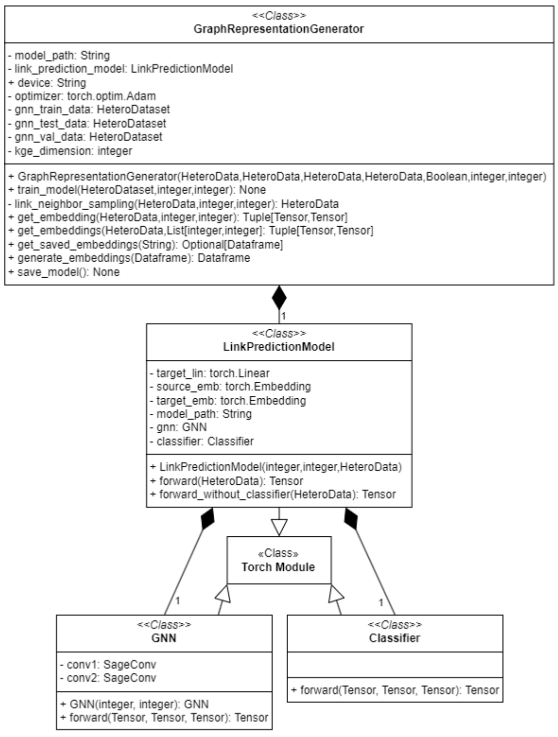

<<Class>>
GraphRepresentationGenerator
- model_path: String
- link_prediction_model: LinkPredictionModel
+ device: String
- optimizer: torch.optim.Adam
- gnn_train_data: HeteroDataset
- gnn_test_data: HeteroDataset
- gnn_val_data: HeteroDataset
- kge_dimension: integer
+ GraphRepresentationGenerator(HeteroData,HeteroData,HeteroData,HeteroData,Boolean,integer,integer)
+ train_model(HeteroDataset,integer,integer): None
- link_neighbor_sampling(HeteroData,integer,integer): HeteroData
+ get_embedding(HeteroData,integer,integer): Tuple[Tensor,Tensor]
+ get_embeddings(HeteroData,List[integer,integer]: Tuple[Tensor,Tensor]
+ get_saved_embeddings(String): Optional[Dataframe]
+ generate_embeddings(Dataframe): Dataframe
+ save_model(): None
1
<<Class>>
LinkPredictionModel
- target_lin: torch.Linear
- source_emb: torch.Embedding
- target_emb: torch.Embedding
- model_path: String
- gnn: GNN
- classifier: Classifier
+ LinkPredictionModel(integer,integer,HeteroData)
+ forward(HeteroData): Tensor
+ forward_without_classifier(HeteroData): Tensor
«Class»
Torch Module
1
1
<<Class>>
GNN
- conv1: SageConv
- conv2: SageConv
+ GNN(integer, integer): GNN
+ forward(Tensor, Tensor, Tensor): Tensor
<<Class>>
Classifier
+ forward(Tensor, Tensor, Tensor): Tensor

Figure 3.6: Class diagram of GraphRepresentationGenerator component

Figure 3.6 shows the class diagram of the component. The LinkPredictionModel, GNN and Classifier are compositions inside the GraphRepresentationGenerator and cannot be accessed from outside otherwise.

GNN class defines a GNN model with two SageConv layers *conv1* and *conv2* and a non-linear ReLU activation function in between, as described in the PyG link prediction tutorial.

Classifier does not store any trainable parameters, but performs the dot product between source and target edge level representations, as described in the PyG link prediction tutorial.

Model is a composition of a GNN and Classifier instance. The forward method first passes all nodes of the given neighborhood HeteroData to the GNN and then pass the KGEs to to classifier, as described in the PyG link prediction tutorial. We add a second forward method (*forward_without_classifier*) to this model, which does the same as *forward*, with the exception, that the KGEs are not passed to Classifier, but returned directly.

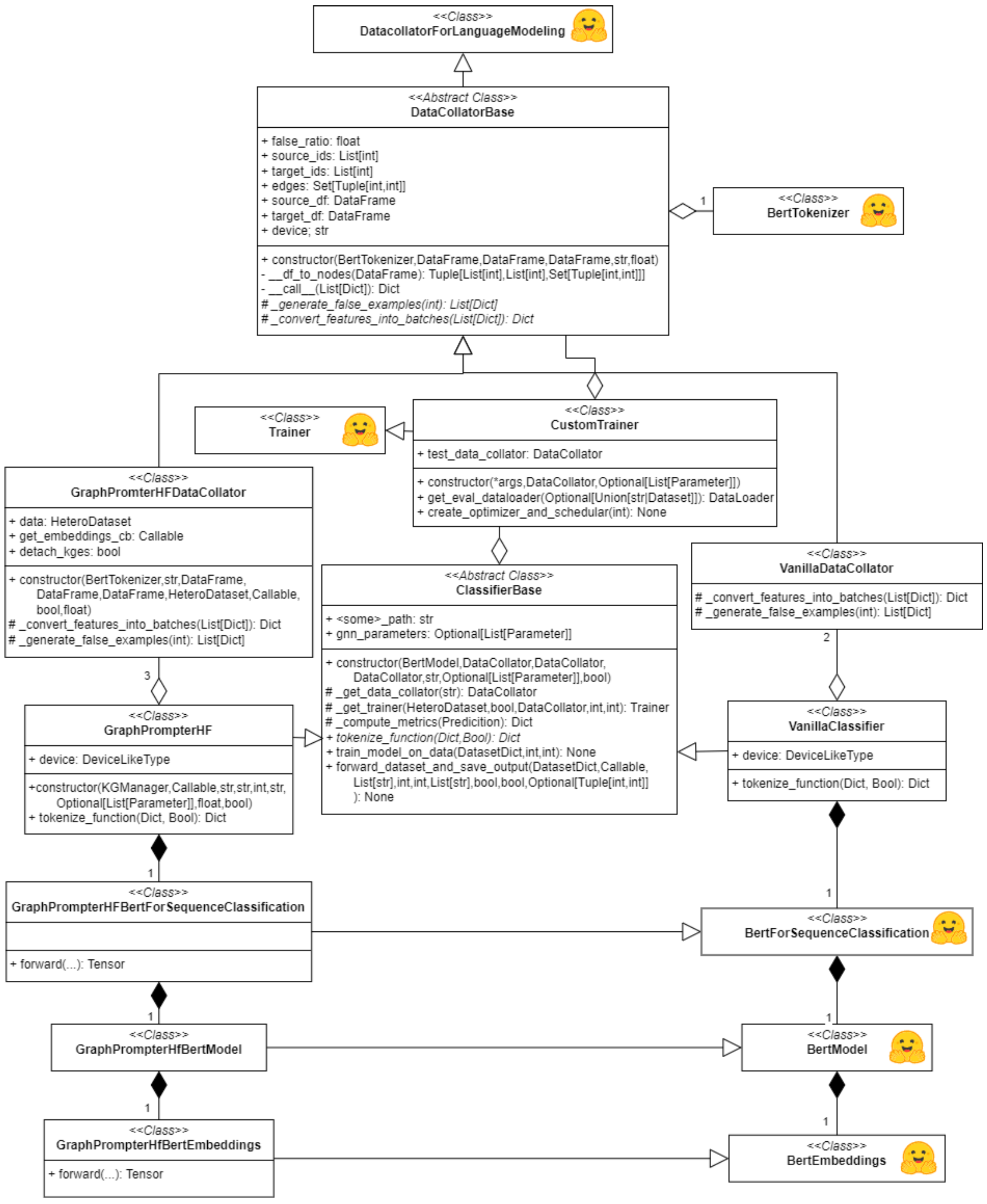
GraphRepresentationGenerator is a the class that provides an interface for other components. The class is initialized with the entire HeteroData and the HeteroData of each split, a flag that can force the class to not load the models weights from disk and the GNN's layer size. The output size needs to be the same as the LLM's hidden state size, so KGEs can be passed directly from one model to another.
The method *train_model* receives a split HeteroData, the number of epochs and batch size and trains the GNN, as described in the PyG link prediction tutorial.
The private method *__link_neighbor_sampling* receives a HeteroData split and a list of source and target ids to return a linked neighborhood HeteroData for each source- target pairs. These neighborhoods can be used to produce KGEs for given source and target ids.
The method *get_embeddings* receives a split HeteroData and a list of source and target pairs and returns KGEs for each of these source- target pairs, by producing linked neighborhoods with *__link_neighbor_sampling* and passing the neighborhoods to the GNN *forward_without_classifier*. The method returns a tuple of tensors, one for source and one for target KGEs. Because the method *__link_neighbor_sampling* is non-deterministic, because the linked neighborhoods are produced by randomly sampling, which may yield different results on every call.
The method *save_model* is used to save the GNN after being trained end-to-end with the LLM.

LLMManager

The LLMManager component manages all LLM related classes, like the Vanilla and GraphPrompterHF models. The component is used for training the models, as well as producing the XAI artifacts, that will be interpreted by ExplainabilityModule. Figure 3.7 illustrates the component diagram of the LLMManager. The LLMManager mostly modifies classes from the Hugging Face transformers library, like BERT, DataCollator and Trainer. Some classes in the component are model independent, like the CustomTrainer and the base classes ClassifierBase and DataCollatorBase. GraphPrompterHF and VanillaClassifier both inherit from ClassiferBase. GraphPrompterHFDataCollator and *VanillaEmbeddingDataCollator* both inherit from the DataCollatorBase. These are the classes that need to be defined for each Training strategy in out case Vanilla and GraphPrompter.

BertForSequenceClassification is a mixture of the Bert base model and a classification header. BertModel produces insert embeddings with the BertEmbeddings class. This class uses multiple trainable feed forward networks to produces *word embeddings* (token embeddings) from *input ids* (token IDs), *position embeddings* from *position ids* (positions) and *token type embeddings* (segment embeddings) from the *token type ids* (segments) and sums them together. The transformation to GraphPrompter for Hugging Face (GraphPrompterHF) is done by inheritance and by overwriting either *forward* and/or *constructor* method. GraphPrompterHFBertEmbeddings method overwrites *forward* to accept *KGEs* and flags for masking the source and target KGEs in addition to *input ids*, *position ids*, *token type ids*. If *KGEs* are passed, the method replaces certain positions in *word embeddings* with *KGEs*, based on *token type ids*, as already described in equation 3.4.
If the flags of masking source and/or target KGEs are passed, the KGE replacement will be skipped, because the mask token has already been replaced before hand. GraphPrompterHfModel simply holds a reference to GraphPrompterHFEmbeddings instead of BertEmbeddings. GraphPrompterHFBertForSequenceClassification holds a reference to GraphPrompterHFBertModel instead of BertModel, accordingly. It also overwrites the forward method, which accepts *KGEs*, *source ids* and *target ids* in addition to *input ids*, *attention mask* and *token type ids*.

ClassifierBase in the center of figure 3.7 is an abstract class and defines what each classifier in this tool suite has to implement. Each *Classifier* is initialized with three DataCollators, one for each split. In addition the model parameters of a PyTorch nn.Module, like of the GNN can be passed, so the trainer can add its parameter to the optimizer during training. The constructor initializes multiple *path parameters* that refer to the file structure we discussed in figure 3.5. A



<<Class>>
DatacollatorForLanguageModeling
<<Abstract Class>>
DataCollatorBase
+ false_ratio: float
+ source_ids: List[int]
+ target_ids: List[int]
+ edges: Set[Tuple[int,int]]
+ source_df: DataFrame
+ target_df: DataFrame
+ device; str
+ constructor(BertTokenizer,DataFrame,DataFrame,DataFrame,str,float)
- __df_to_nodes(DataFrame): Tuple[List[int],List[int],Set[Tuple[int,int]]]
- __call__(List[Dict]): Dict
# _generate_false_examples(int): List[Dict]
# _convert_features_into_batches(List[Dict]): Dict
1
<<Class>>
BertTokenizer
<<Class>>
Trainer
<<Class>>
CustomTrainer
+ test_data_collator: DataCollator
+ constructor(*args,DataCollator,Optional[List[Parameter]])
+ get_eval_dataloader(Optional[Union[str|Dataset]]): DataLoader
+ create_optimizer_and_schedular(int): None
<<Class>>
GraphPromterHFDataCollator
+ data: HeteroDataset
+ get_embeddings_cb: Callable
+ detach_kges: bool
+ constructor(BertTokenizer,str,DataFrame,
DataFrame,DataFrame,HeteroDataset,Callable,
bool,float)
# _convert_features_into_batches(List[Dict]): Dict
# _generate_false_examples(int): List[Dict]
<<Abstract Class>>
ClassifierBase
+ <some>_path: str
+ gnn_parameters: Optional[List[Parameter]]
+ constructor(BertModel,DataCollator,DataCollator,
DataCollator,str,Optional[List[Parameter]],bool)
# _get_data_collator(str): DataCollator
# _get_trainer(HeteroDataset,bool,DataCollator,int,int): Trainer
# _compute_metrics(Predicition): Dict
+ tokenize_function(Dict,Bool): Dict
+ train_model_on_data(DatasetDict,int,int): None
+ forward_dataset_and_save_output(DatasetDict,Callable,
List[str],int,int,List[str],bool,bool,Optional[Tuple[int,int]]
): None
<<Class>>
VanillaDataCollator
# _convert_features_into_batches(List[Dict]): Dict
# _generate_false_examples(int): List[Dict]
2
3
<<Class>>
GraphPrompterHF
+ device: DeviceLikeType
+constructor(KGManager,Callable,str,str,int,str,
Optional[List[Parameter]],float,bool)
+ tokenize_function(Dict, Bool): Dict
<<Class>>
VanillaClassifier
+ device: DeviceLikeType
+ tokenize_function(Dict, Bool): Dict
1
<<Class>>
GraphPrompterHFBertForSequenceClassification
+ forward(...): Tensor
1
<<Class>>
BertForSequenceClassification
1
<<Class>>
GraphPrompterHfBertModel
1
<<Class>>
BertModel
1
<<Class>>
GraphPrompterHfBertEmbeddings
+ forward(...): Tensor
1
<<Class>>
BertEmbeddings

Figure 3.7: Class diagram of LLMManager component. Classes of the original Hugging Face transformers library as in *Transformers: State-of-the-Art Natural Language Processing*, by Wolf et al., 2020, are labeled with the Hugging Face logo

classifier can again be forced to not load a fine tuned model from disk, via the flag.
The protected method *_get_trainer(str)* receives a *Hugging Face dataset*, a, the amount of epochs for training and batch size. The method returns a *Trainer* instance.
The method *train_model_on_data* receives a *Hugging Face DatasetDict* with all splits, the amount of epochs and the batch size and initialized the training on given *DatasetDict*.
Each implementing class of *ClassiferBase* has to implement the method *tokenize_function*. This method accepts a *Dictionary* of features, like *prompt*, *source* and *target ids* and *labels* and it accepts a flag if the resulting Dict contains *PyTorch Tensors* or Python base types. The method tokenizes every prompt to it's *input ids* and *attention mask*. Because each prompt follows the same structure, the method also generates the *token type ids*. Then the method returns a Dictionary of *input ids*, *attention mask*, *labels*, *token type ids* and in the case of *GraphPrompterHF* the *source* and *target ids*.

The most complex method is *forward_dataset_and_save_output* is the method that produces all data used by the *ExplainabilityModule*. We call the outputs of this method **XAI artifacts**, because they contain rich and memory efficient data about the models behavior, that can be interpreted by explainability tools and algorithms.
The method receives a Hugging Face DatasetDict with the *Dataset* splits, a list of strings, with the split names we want this method to produce XAI artifacts for, the batch size during inference, the amount of batches that are forwarded, before intermediate steps of the XAI artifacts are saved, a list of XAI artifact names, that are to be produced and saved (can be "attentions", "hidden_states" and/or "logits"), an optional tuple of boundaries, that are used for horizontal scaling on multiple machines and a flag if the computation of XAI artifacts can be skipped if they were found on disk. Later we look more into detail of the XAI artifact process.

VanillaClassifier inherits from *ClassifierBase*, implements the *tokenize_function* method and initializes the VanillaDataCollator and VanillaBertForSequenceClassification model correctly. The constructor receives three DataFrames, one with all LLM related data, one with only *source ids* and one with only *target ids*. These are passed to the DataCollators, so they can produce non-existing source-target pairs for the training process. The constructor also receives a path to the root directory, the model name in the Hugging Face Hub, the model max length, a ratio of how many non existing edges are to be produced on the fly and the flag to disable loading a fine tuned model from disk. The constructor initializes the tokenizer and two DataCollators. One DataCollator is for training and has a false ratio between zero and one, while the other does have a false ratio of minus one. This makes the DataCollators behave differently when it comes

to generating non-existing source-target data points during training. The constructor then loads configs for the BERT model. The configs *type vocab size* are increased, so the BertModel will accepts four token type ids (segments). Then the *VanillaBertForSequenceClassification* instance is loaded from disk or from the Hugging Face Hub of pre-trained BERT models.
The VanillaClassifier also implements the *tokenize_function*, as we discussed in the Classifier-Base.

GraphPrompterHF inherits from *ClassifierBase* and also implements the *tokenize_function*. This time the constructor initializes three DataCollators. Not only differ the DataCollators in their false ratio, but also which HeteroData they are using for the KGE generation. We talk about this more in detail when explaining the GraphPrompterHFDataCollator.
The constructor receives the entire KGManager instead of only the DataFrames, but essentially only accesses them and other dataset formats from the KGManager. Two additional parameters can be passed, namely a Vanilla model path, so the BERT model can be trained upon the fine-tuned Vanilla model and GNN model parameters, which allows the *Trainer* to add these parameters to the optimizer, which enables the end-to-end training of the Bert Model with the GNN.
The implementation of *tokenize_function* only differs from the Vanilla model by the token types expected in the prompt.

DataCollatorBase is an abstract class that inherits from the Hugging Face's DataCollator-ForLanguageModeling. DataCollatorBase holds reference to a float *false_ratio*, which tells the DataCollator how many incoming datapoints are to be replaced with node-pairs, that do not share an edge with each other. The DataCollator also holds references to all available source and target node ids, all edges between those, as well as Pandas DataFrames with all the features of those nodes. Lastly the class holds a reference to the device the LLMs are working on.
DataCollatorBase overwrites the private method *__call__* from the DataCollatorForLanguage-Modeling class. Instead of forwarding given data points directly to the protected method *_convert_features_into_batches*, *__call__* will replace a fixed amount of them with data points that represent source and target nodes, that do not share an edge between each other. For that it calls the protected method *_generate_false_examples* on a percentage of data points defined in the *false_ratio*. If this ratio is *-1*, the DataCollator will skip the replacement process.
DataCollatorBase's constructor accepts a Hugging Face tokenizer, the Pandas DataFrames from the DatasetManager with the entire dataset, with the source nodes and with the target nodes, as well as the device and the *false_ratio*. The DataFrame with the entire dataset is transformed

into three lists of integer, each representing all source node ids, all target node ids and all edges between these nodes. The DataCollator uses these lists to produce node-pairs that do not exist in the graph and that are not part of the other splits.
Each implementing class of DataCollatorBase needs to implement the protected methods *_generate_false_examples* and *_convert_features_into_batches*.

VanillaDataCollator implements the abstract class DataCollatorBase and with that the said protected methods.
The constructor of this class is the very same as of DataCollatorBase and only propagates it's parameters.
The method *_generate_false_examples* accepts an integer, indicating how many non-existing node pairs are to be generated. For each node-pair generated, the method also generates the input prompts, then tokenizes them and generates the *_attention_mask* and *token_type_ids*, as we already discussed in the *tokenize_function*. The resulting list of dictionaries are passed back, replacing existing node-pair datapoints. The entire list is then passed to *_convert_features_into_batches*.
The protected method *_convert_features_into_batches* accepts a list of dictionaries in which we expect to find *input_id*, *attention_mask*, *labels* and *token_type_ids*. Each of these features are grouped together in PyTorch tensors. The method then returns a dictionary of Pytorch tensors.

GraphPrompterHFDataCollator is initialized in addition to the parameters of the VanillaDataCollator with a string representing the split this DataCollator is responsible for. This label is later passed to the forward method of GraphPrompterHF *forward* method, so when this method generates KGEs, it does know, from which split (train, test or validation) the linked neighborhood can be produced from.
The method *_generate_false_examples* returns in addition to the *input ids*, *attention mask*, *labels* and *token type ids* also the *source id* and *target id* of the newly generated node-pair.
The method *_convert_features_into_batches* receives all said parameters and *source-* and *target-KGEs*. These two parameters can be used during evaluation mode after training, if there is no need to generate KGEs for every masked forward pass iteration individually.

CustomTrainer has two additional responsibilities during training. The first one is, that there are two separate DataCollators, one for training and one for evaluation. Again, this allows the evaluation process to use node neighborhoods from a separate pool. Both DataCollators are passed during initialization. We also overwrite the *get_eval_dataloader*, so that this method return the evaluation DataCollator if available.

The other task of this Trainer is to add the GNN model parameters to the optimizer. If said parameters are passed during the initialization process, the overwritten method *create_optimizer_and_scheduler* will add the given parameters to the optimizer.

ExplainabilityModule

The ExplainabilityModule only consists of one class. This class is responsible for producing plots of the XAI artifacts, like the accuracy during the training process, confusion maps, SHAP-values (Lundberg and Lee [2017]), attention maps, PCA dimensionality reduced hidden states and some other model features.

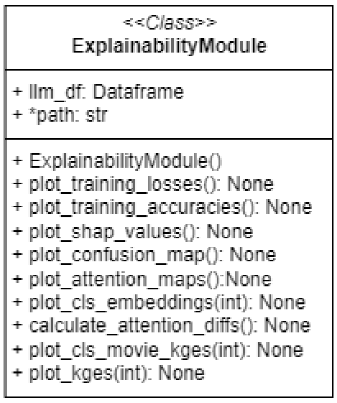

<<Class>>
ExplainabilityModule
+ llm_df: Dataframe
+ *path: str
+ ExplainabilityModule()
+ plot_training_losses(): None
+ plot_training_accuracies(): None
+ plot_shap_values(): None
+ plot_confusion_map(): None
+ plot_attention_maps():None
+ plot_cls_embeddings(int): None
+ calculate_attention_diffs(): None
+ plot_cls_movie_kges(int): None
+ plot_kges(int): None

Figure 3.8: ExplainabilityModule component

Figure 3.8 shows the class diagram of the ExplainabilityModule. The class holds a reference to the entire dataset and multiple paths to all XAI artifacts produced in previous steps. The *constructor* of this class loads the dataset and initializes all paths.
The method *plot_training_losses* plots all training losses in a coherent manner during training of the GNN, Vanilla, GraphPrompterHF frozen and GraphPrompterHF end-to-end models. Because the training started with the Vanilla and GNN models, continued with GraphPrompterHF frozen then with the end-to-end version, the losses and accuracy are placed on the diagram the same way, resulting in a continues loss graph. A flag can be passed, if the plot is not to be saved on disk.
Method *plot_training_accuracies* behaves the exact same way, with the difference, that the accuracy during training is plotted.
Method *plot_shap_values* computes the SHAP-values (Lundberg and Lee [2017]) with equation 2.1 for all models and all segments.
Method *plot_attention_map* visualizes the attention scores of all models as discussed in section

3.0.2.
Method *plot_cls_embeddings* takes the hidden states of the classifier token *[CLS]*, reduces it's dimensions via PCA to two dimensions and visualizing the resulting vector representations of given amount of samples on a scatter plot via Matplotlib (Hunter [2007]). In the resulting two plots we color and mark all 2D points depending on their ground truth.
Method *plot_kges* takes the hidden states of user and movie KGEs in the initial layers of both GraphPrompterHF models, computes the average cosine similarity (normalized dot-product) between user and movie KGEs and prints them on the output stream. Then the hidden states dimensions are reduced to 2D points and visualized on a scatter plot with one coloring and marking setup.
Method *plot_cls_movie_kges* works different, because this method takes the hidden states of XAI artifacts, that were produced for computing SHAP-values. The XAI artifacts in this method are coming from the forward passes, in which only classification token and KGEs were not masked. This method takes the hidden states of the classification token in the input layer and the user and movie KGEs in the input and first layer of both GraphPrompterHF models, reduces their dimensions via PCA and visualizes them on a scatter plot. This is done in three different color and marker setups. Then this method also computes the average attention score for all discrete values of movie connectivity between classifier token in the input layer and movie KGE in the first layer and plots it as a Matplotlib line plot. Last, this method computes the average difference between the cosine similarity of KGEs in the input layer compared to the first layer.

3.1.3 Process View

We have discussed the key concepts and static architecture of the tool suite. Now we are looking at the multiple pipeline steps during run time. Each pipeline step is implemented in Jupyter Notebooks (Loizides and Schmidt [2016]). We will look at each notebook and explain the sequential interactions between the classes, we have described in the previous section.

Initialize Dataset and Train GNN

The notebook *preprocess_dataset_train_gnn.ipynb* loads the MovieLens dataset, transforms it into the standardized format and then trains the GNN on its data.

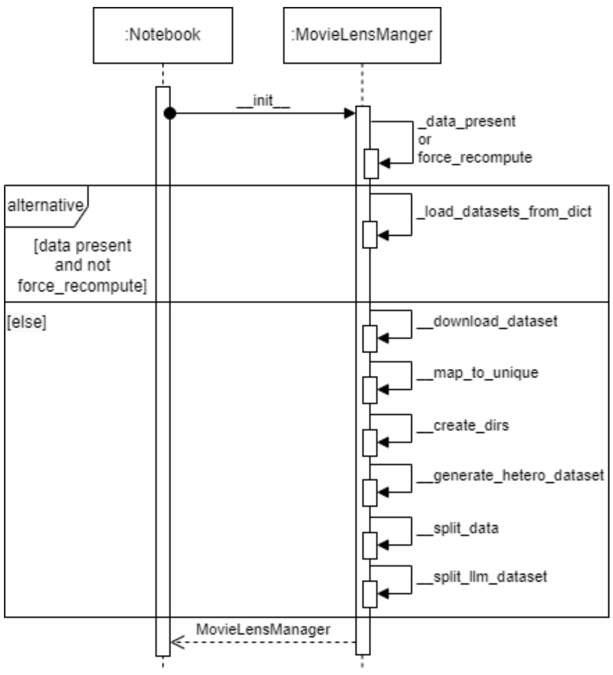
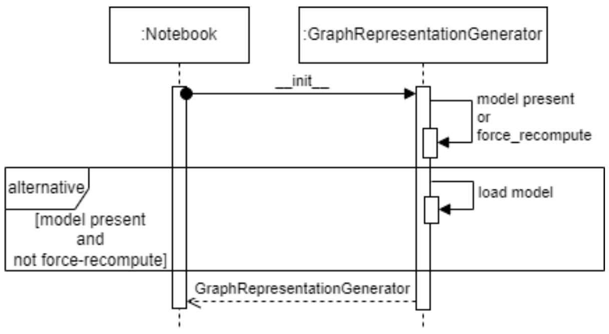
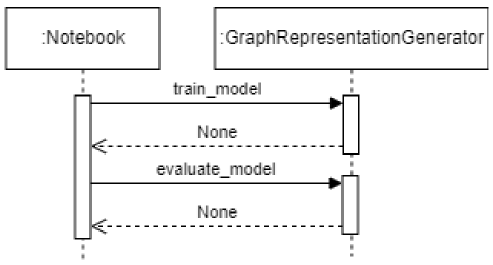


Figure 3.9: MovieLensManager initialization

Figure 3.9 shows the initialization process of the *MovieLensManager*. The constructor checks at the beginning of each initialization process, if the dataset has already been initialized before. If so, the *MovieLensManager* loads the dataset from disk. If not, than the constructor loads the original dataset and transforms it. Every following step assumes, that the MovieLensManager has already been initialized ones and we will refrain from showing this process in the following diagrams.



(a) GraphRepresentationGenerator initialization

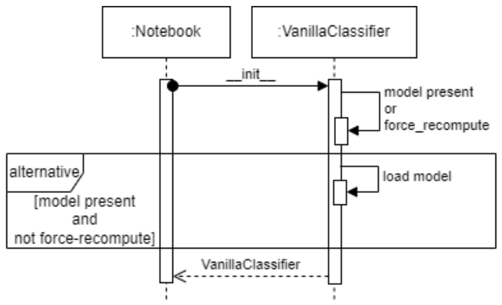


(b) GNN training and evaluation

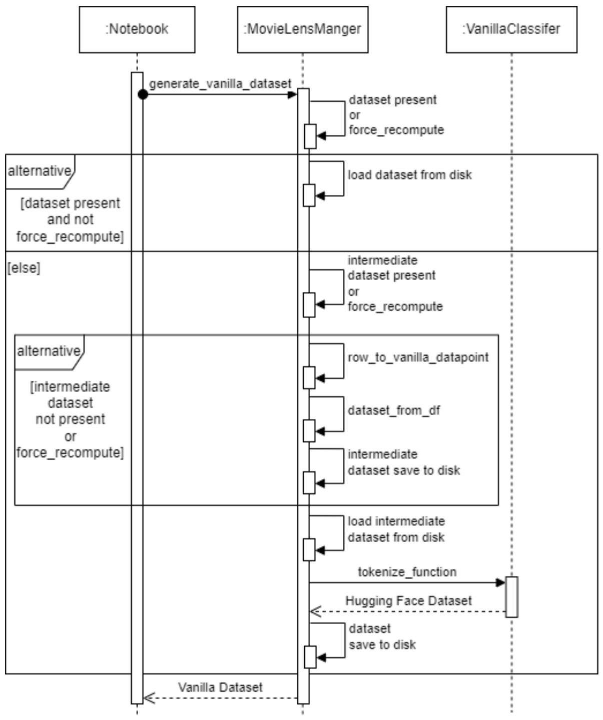
Figure 3.10: GraphRepresentationGenerator initialization and GNN training and evaluation

Figure 3.10a shows the initialization process of the *GraphRepresentationGenerator* class. The constructor loads the model for the GNN if one is saved at the expected relative path. The initialization of *MovieLensManager* and *GraphRepresentationGenerator* follows the training and evaluation of the GNN model, as depicted in figure 3.10b

Vanilla Dataset Generation and Model Training



(a) VanillaClassifier initialization


:Notebook
:MovieLensManger
:VanillaClassifer
generate_vanilla_dataset
dataset present or force_recompute
alternative
[dataset present and not force_recompute]
load dataset from disk
[else]
intermediate dataset present or force_recompute
alternative
[intermediate dataset not present or force_recompute]
row_to_vanilla_datapoint
dataset_from_df
intermediate dataset save to disk
load intermediate dataset from disk
tokenize_function
Hugging Face Dataset
dataset save to disk
Vanilla Dataset

(b) Vanilla dataset generation

Figure 3.11: Vanilla model initialization and Vanilla dataset generation

The notebook *training_vanilla_model.ipynb* initializes the *Vanilla dataset* and trains the *Vanilla-Classifier*. Figure 3.11a shows the same concept of loading the LLM if available from disk, as with the *GraphRepresentationGenerator*. The beginning of *generate_vanilla_dataset* shown in figure 3.11b, works the same way, as with the other classes. If the dataset has already been generated, the dataset can be loaded from disk. In case the dataset has not been loaded, we want to load an intermediate dataset from disk, that has not yet been passed to the tokenizer. The reason for that behavior is, that the method *tokenize_function* does behave differently, if the given dataset was loaded from disk or was entirely loaded from memory. If latter, the tokenizer can lead to memory overflows. If the intermediate dataset was also not found on disk, it is generated and then saved to disk. Right after the dataset generation, the training of the *VanillaClassifier* begins.

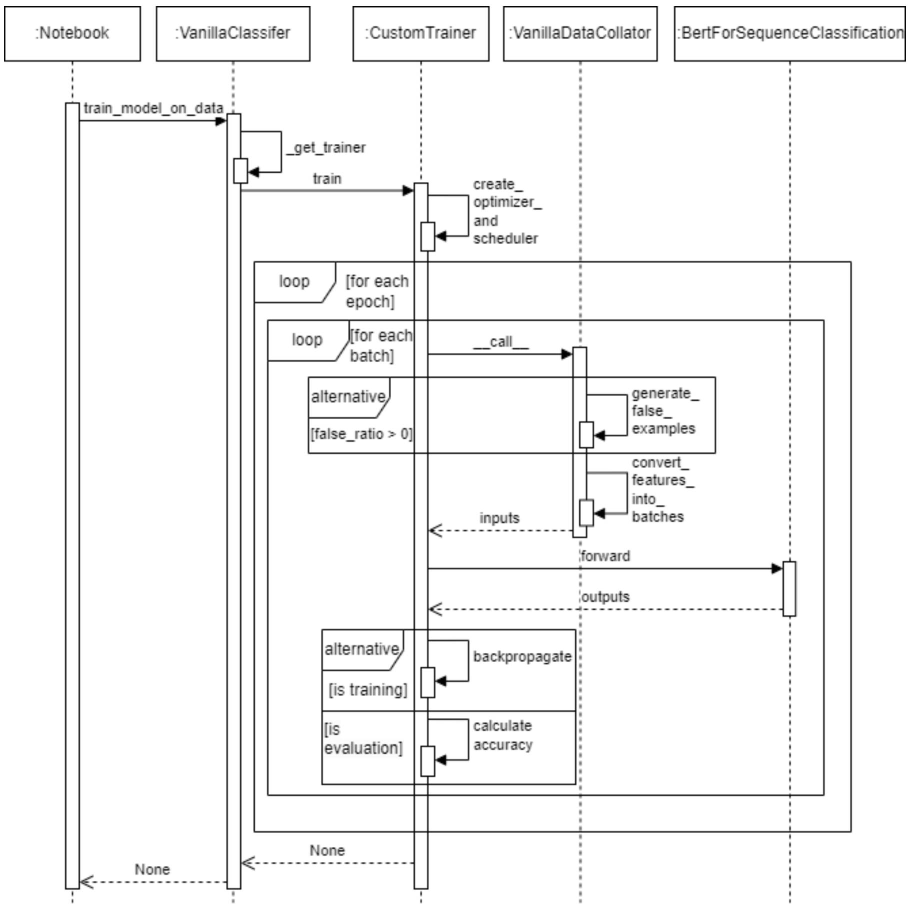


Figure 3.12: Training Vanilla model

Figure 3.12 shows the training process of the Vanilla model. First the *CustomTrainer* gets generated and the training process gets started. The trainer calls the DataCollator for every step in every epoch and have that DataCollator generate batches of inputs for the model. The DataCollator has two modes, in which it generates input batches for the model. If the DataCollator was initialized with a *false_ratio* between zero and one, then datapoints of node pairs are generated, that do not share an edge and they replace some or all of the datapoints in the current batch. Then each data point is formatted into *PyTorch tensors* with the method *convert_features_into_batches*.

Then the data points are passed to the *Bert* model. During training, the loss is calculated and propagated back, to train the model. During evaluation the accuracy in calculated instead.

GraphPrompterHF Dataset Generation and Model Training

The generation of the *GraphPrompterHF* training dataset is done the same way as with the *Vanilla* model, with the difference in how the prompts look like.
The training is also mostly the same, with a big difference in the way the classifier forwards the input data. We will refer to figure 3.12 for the entire training process and go more into detail for *forward* method call of *GraphPrompterHFBertForSequenceClassification*.

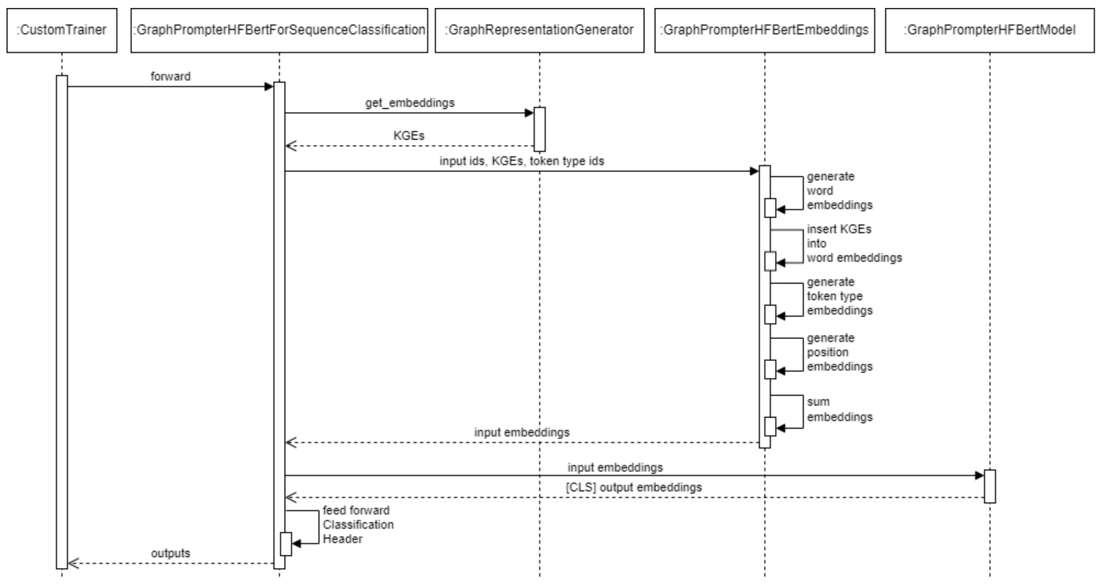

:CustomTrainer
:GraphPrompterHFBertForSequenceClassification
:GraphRepresentationGenerator
:GraphPrompterHFBertEmbeddings
:GraphPrompterHFBertModel
forward
get_embeddings
KGEs
input ids, KGEs, token type ids
generate word embeddings
insert KGEs into word embeddings
generate token type embeddings
generate position embeddings
sum embeddings
input embeddings
input embeddings
[CLS] output embeddings
feed forward Classification Header
outputs

Figure 3.13: Training GraphPrompterHF model

Figure 3.13 shows the feed forward process of the *GraphPrompterHF* model. First *GraphPrompterHFBertForSequenceClassification* calls the callback function of the *GraphRepresentationGenerator* to produce KGEs. These KGEs, input ids and token type aids are then passed to *GraphPrompterHFBertEmbeddings*. There the word embeddings are produced by forwarding the input ids. Two of these embedding positions are then replaced with the source and target KGEs. Then the input embeddings are produced by summing the word embeddings with the token type embeddings and position embeddings. The resulting input embeddings are then passed to *GraphPrompterHFBertModel*, where the input embeddings are passed through the multi-head attention layers. The output embeddings of the classification token *[CLS]* are then passed to the classification header for the actual prediction.

Generate XAI Artifacts

The generation of the data points, that reflect the models behavior, like the generation of attention maps, hidden states and logits for the SHAP-value calculation happens in the *Jupyter Notebooks forward_data_vanilla*, *forward_data_graph_prompter_hf* and *forward_data_graph_prompter*. Each notebook starts the *forward_dataset_and_save_outputs* with specific settings.

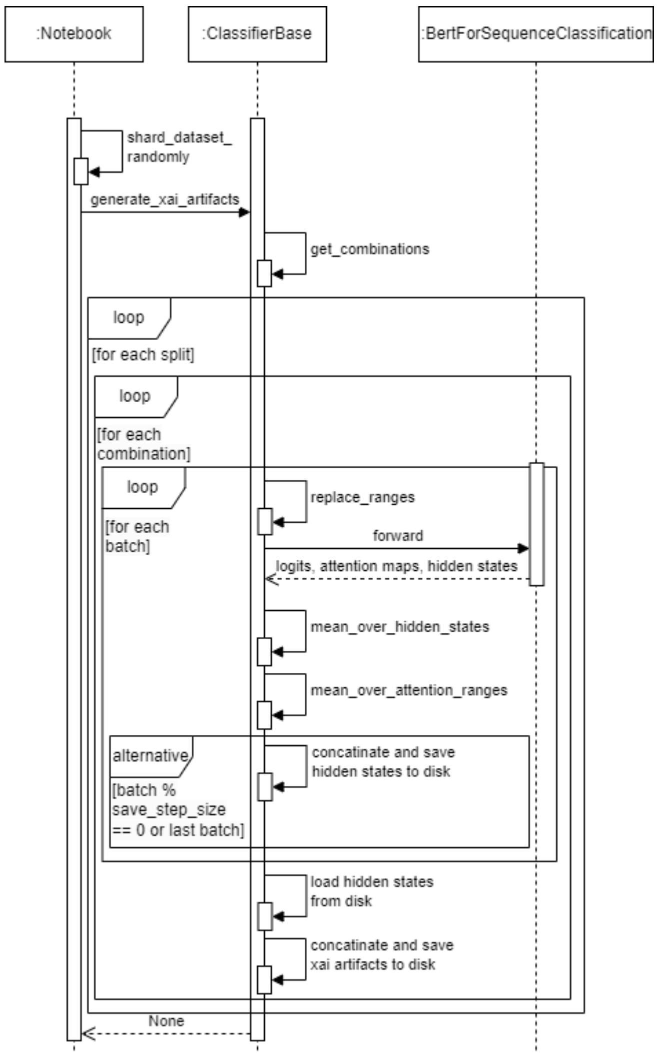

:Notebook
:ClassifierBase
:BertForSequenceClassification
shard_dataset_ randomly
generate_xai_artifacts
get_combinations
loop
[for each split]
loop
[for each combination]
loop
[for each batch]
replace_ranges
forward
logits, attention maps, hidden states
mean_over_hidden_states
mean_over_attention_ranges
alternative
[batch % save_step_size == 0 or last batch]
concatinate and save hidden states to disk
load hidden states from disk
concatinate and save xai artifacts to disk
None

Figure 3.14: Generate XAI artifacts by forwarding masked data points

Figure 3.14 shows the process of generating XAI artifacts that is the same for each model. First we sample a fixed amount of data points from the dataset, so we do not have to compute the artifacts for all data points. Next, all combinations of token type id masks in the given dataset are generated. The dataset is then processed ones for each combination and split. In each batch iteration, the attention mask is masked with the token type ids and the current combination of token type masks. The Bert models then produce the XAI artifacts logits, attention maps and hidden states. Hidden states and attention maps are then grouped by they token types. Because the hidden states may result into memory overflows, they are saved from time to time on disk, while all of the XAI artifacts are concatenated and saved on disk each dataset split and combination.

3.1.4 Implementation Tests of Segment Equations

We have tested the correctness of implementations that could introduce unnoticed errors, like the equations that use the segments (token type ids) for averaging or replacing. Replacing the padding tokens with KGEs and segment groups with mask tokens was tested manually, printing the output token type id for some forward passes and checking the correct structure. The implementations of equations for grouping hidden states and attention scores by segments were tested by implementing a less performance oriented but human readable form of the equation. These tests were deemed to have been passed, if the outputs of the original implementation and the slower readable implementation return the same outputs, whereby a rounding error was permitted.

4 Experiments

Our goal is to find out, what internal semantic behavior our GraphPrompterHF architecture on the task of link prediction on the MovieLens dataset (Harper and Konstan [2015]) shows up using XAI methods. The experiment is divided into multiple stages, each producing outputs that are necessary for following stages. In the first stage, we load the MovieLens dataset from its source[1], preprocess it for training and then train the models on the task of link prediction. Next, we are producing XAI artifacts with the trained models. Finally, we are evaluating the given XAI artifacts with XAI methods.

4.1 Producing XAI Artifact

4.1.1 Dataset Preprocessing

The MovieLens dataset (Harper and Konstan [2015]) contains 32 million movie ratings with 87.585 movies and 200.948 users. The dataset is transformed into a *HeteroData* dataset. The dataset's source nodes represent the users, while the target nodes represent the movies. The feature *genres* of the movie are also transformed into one hot encoding and part of the movie nodes.
We split the dataset with a ratio of $(80, 10, 10)$, so the training data contain eighty percent of the entire dataset. The training data set is split again into two parts with the ratio of $(70, 30)$, so that 70 percent of the training data will be used only for message passing, while the rest 30 percent are used for the training.
Every dataset split is then doubled in size by adding false edges for every existing edges. For the training split this is done on the fly, while for the datapoints for test and validation splits are fixed.
With the *HeteroData* datasets we generate the DataFrames for the LLM training, in which the user column only contains the *user id* and the movie columns contain the *movie id*, *title* as a string and *genres* as a list of strings.
We manually test the NLP datasets for disjunct datapoints, so there can be no information leak.

[1]URL to the MovieLens dataset 32 million datapoints: https://files.grouplens.org/datasets/movielens/ml-32m.zip

4.1.2 Model Training

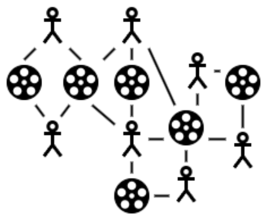
The models are trained on the datasets provided in the previous stage. First the GNN and Vanilla model are trained, then the *GraphPrompterHF* frozen model and last the *GraphPrompterHF* end-to-end model.
We train the GNN model for twenty epochs with a twenty neighborhood size on the first hop and ten neighborhood size on the second hop with a false-edge ratio of $(50, 50)$.
The dataset splits are transformed into Hugging Face datasets with the columns *prompt* and *labels*. A *prompt* has the structure of $user\_id[SEP]movie\_id[SEP]title[SEP]genres$. Each datapoint is passed to the BERT tokenizer, which adds the classifier token ($[CLS]$) at the beginning and an additional separator token (*[SEP]*) at the end of the prompt. The tokenizer calculates the attention mask based on the sequence length and token type ids based on the separator token ids and adds them to the output. This results in a dataset that has the columns *input_ids*, *labels*, *attention mask* and *token_type_ids*.
The Vanilla model is initialized on the $BERTTiny_{128\_2}$ version and trained for two episodes on the train split, with a batch size of 256, warm up steps of 500 and weight decay of 0.01. Each epoch the training process is validated against the dataset test split.
The dataset splits are again transformed Hugging Face datasets with the columns *prompt* and *labels*. This time, the *prompt* structure is
$user\_id[SEP]movie\_id[SEP]title[SEP]genres[SEP][PAD][SEP][PAD]$. The rest of the tokenization process is the same as with the Vanilla model.
The GraphPrompterHF frozen model is initialized on the Vanilla model but during initialization, we do not pass the GNN model parameters, so that GNN stays frozen during training. The model is trained on the same hyper parameters as before.
The training process on the end-to-end version of GraphPrompterHF uses the same dataset as for the frozen version. But this time during initialization, we are also passing the GNN model parameters to the GraphPrompterHF, so that all models can be trained end-to-end.

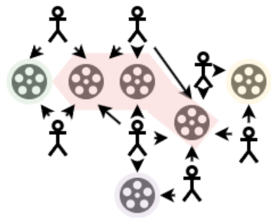
4.1.3 Forward Dataset to Models

We sample from the *Vanilla* and *GraphPrompterHF* datasets approximately 100000 datapoints from the test and validation split and forward them to the models. For each combination of segment groups, we collect the logits, hidden states and attention maps, reduce their precision to four decimal points and store them on disk.
Then we are fusing the shards of hidden states together, so there is one Numpy array for each model, dataset split, and segment group. The XAI artifacts are now ready for evaluation.

4.2 XAI Evaluation

In this section we produce plots and extract quantized features of the XAI artifacts. Our expectations are as followed:
The Vanilla model will perform worst, followed by GraphPrompterHF frozen, followed by GraphPrompterHF end-to-end followed by the GNN alone. We expect this behavior, because this task was chosen in a way, that favors structured based predicting, which the GNN is most sophisticated in. The GraphPrompterHF models should follow, because they simply carry more information into the prediction, then the Vanilla model. Frozen GraphPrompterHF should fall behind end-to-end GraphPrompterHF, because the end-to-end model has more parameters to remember the user-movie relations.
We are expecting to see some unusual behavior of GraphPrompterHF dealing with the KGEs injected via soft prompts. Because these KGEs were generated in a way, that the similarity between them indicates the likelihood of an edge between them, the transformer will have to leverage that similarity between two KGEs for interpretation. Because similarity is only part of the attention score process, we expect GraphPrompterHF to transform these scores into actual vector representations.
We expect the process of adding false-edges to the dataset for data points, the way it was conducted in this project, as the source of an unintentional bias. We expect the movies of MovieLens dataset to be different in popularity (degree of incoming edges to the movie node). If we add false-edge data points to the dataset uniformly, we are introducing a bias, that overrepresents negative nodes of movies, that are not popular. This process should take the movie and user popularity into consideration, so that the distribution of incoming and outcoming edges of nodes stay the same.


(a) Example Graph of user-movie relations.


(b) Expected reasoning of Vanilla model

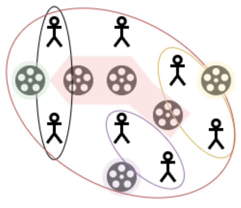
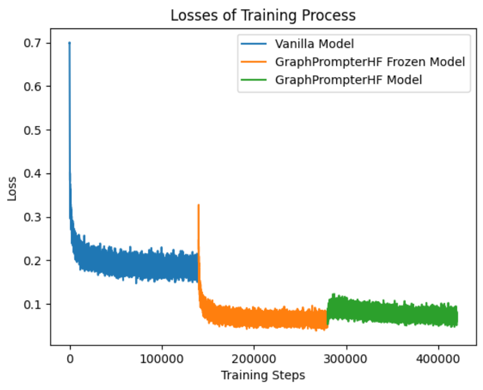

(c) Expected reasoning of GraphPrompterHF models

Figure 4.1: Expected reasoning of models over example graph of users (mannequins) and movies (film rolls) via learned clusters (bubbles and outlines).

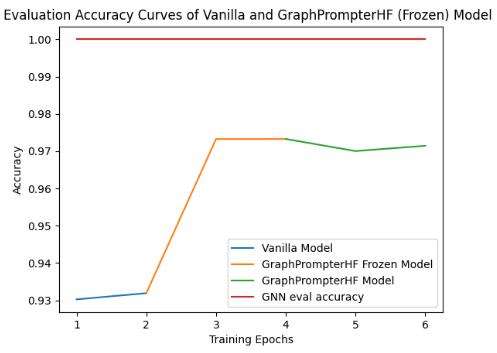
Figures 4.1 illustrate the expected reasoning behavior of the Vanilla and GraphPrompterHF models. In figure 4.1a we illustrate a possible subgraph of the MovieLens dataset. In this graph all users are connected to one or more movies. All movies are connected to one more users. There are movies that are only connected to two users, while there are movies that are connected to four or even five users. We expect the Vanilla model to leverage features of movies for generalizing movies into groups and remember the connection between arbitrary user IDs and movie groups as illustrated in figure 4.1b. For the GraphPrompterHF we expect them to also generalize user groups and learn the connection between user and movie groups, which should generalize better.

4.2.1 Model Performances

The first view we are looking at is the models performance during training.



(a) Training Losses of the Vanilla and GraphPrompterHF models



(b) Training Accuracies of all (GNN, Vanilla, GraphPrompterHF) models

Figure 4.2: The performance of all models during training

In figure 4.2 we can see the performance evolution during training. The losses in 4.2a show a steady and quick decrease in the Vanilla model training, in which it fluctuates around 0.2 till the end. The loss of the frozen GraphPrompterHF model spikes at the beginning of training and then decreases quickly, in which it fluctuates around 0.075. The loss of the end-to-end GraphPrompterHF model increases at the beginning and stays around 0.9 until the end.
Figure 4.2b shows the accuracy development during training of the models. The GNN shows an accuracy of 1.0, which stays so during the end-to-end training. The Vanilla model shows an accuracy of 0.93 in the first epoch and an increase in the second epoch to 0.932. The accuracy during the training of the frozen GraphPrompterHF model increases in the first epoch to 0.973,

where it stays till the end. The accuracy of end-to-end GraphPrompterHF model decreases in the first epoch to 0.97 and rises again to 0.971 in the second epoch.

The models performance in the figures 4.2 do not meet our expectations. First we have an overfitting GNN model, that performs with an accuracy of 1.0. This overfitting should be invested into, as it influences our interpretation of the models behaviors.
Another unexpected behavior comes in the order of GraphPrompterHF versions. We expected the end-to-end model to perform better then the frozen version. But there is a slight decrease in accuracy and a slight increase in loss the other way around. So far we can only make speculations about this behavior. The overfitting of the GNN may have been the result of too many training epochs. The accuracy drop may have been the result of the use of the PyG *RandomLinkSplit* with *disjoint_train_ration*. This ratio reserves some nodes for message passing across all dataset splits instead of training. This ratio was set to 70%, so more then half of the entire dataset's nodes were used only for message. During the translation from a HeteroDataset to a Hugging Face dataset, we unintentionally ignored this behavior and assigned all datapoints for message passing to the training split. The GraphPrompterHF versions had a greater amount of data points to train on. With the greater amount and the overfitting, the models lost some of their generalizability. Then during the end-to-end training, the overfitting of the new data points did not influence the GNNs to the point where it dropped in accuracy, but it probably lost some generalizability.

4.2.2 SHAP-values

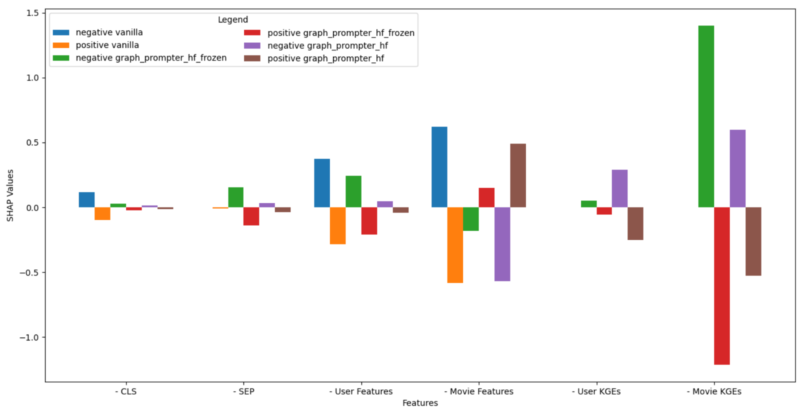


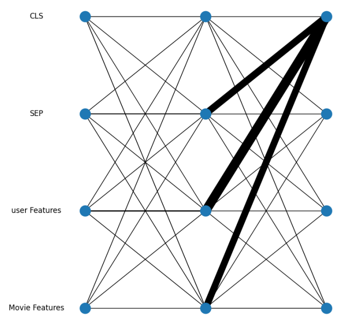
Figure 4.3: SHAP-values of models grouped by token types

Figure 4.3 shows the SHAP-values of each model for each token type (segment). The bars are ordered from negative to positive influences for each model. The figure shows, that the influence by the classification token is overall small, while the influence is highest in the Vanilla model. The separation tokens also have a relatively low influence on all models, tho they show a relative higher influence on the frozen GraphPrompterHF model. The user features show moderate influences on the Vanilla and frozen GraphPrompterHF model, but low influence on the end-to-end GraphPrompterHF model. The movie features show high influence on the Vanilla and end-to-end GraphPrompterHF, but low influence on the frozen GraphPrompterHF model. The influences of KGEs are only shown for the GraphPrompterHF models, because the Vanilla model does not use these features. The user KGEs do show low influence on the frozen GraphPrompterHF model, but moderate influence on the end-to-end version. The movie KGEs show a very high influence on the frozen version and moderate influence on the end-to-end model version.

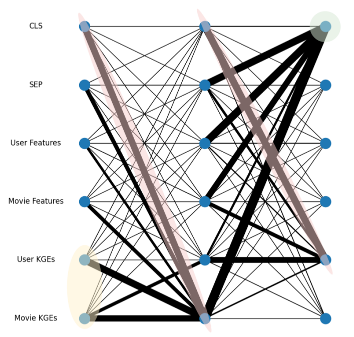
It seems, that the models pay high attentions to the movies, because the user features do not contribute much to the result, as the user ID is arbitrary and carries no semantic meaning. That leaves the *Vanilla* model with two strategies. The first one is to remember each individual user and its movie preferences and the second one is learning the overall most popular movie features. The high influence of movie features in the Vanilla model could be the result of the transformer learning popular movie features.
The frozen GraphPrompterHF pays the most attention by far to the movie KGEs and almost no attention to the user KGEs, but relatively high attention to the user features. This behavior shows that the movie KGEs holds crucial information for the given down stream task. The moderate influence of the user features looks similar to that of the Vanilla model's behavior. This could mean, that the frozen GraphPrompterHF model continues the prediction strategy of the Vanilla model but using the richer movie KGEs instead. A question arises, why the frozen GraphPrompterHF does not also rely on the user KGEs. We can certainly say, that the transformer did not pick up the similarity approach on user and movie KGEs. Else the importance of user KGEs would be much higher. If the Vanilla model actually tried to learn the preferences of each user individually, this could explain the frozen GraphPrompterHF's behavior for not switching to the user KGEs.
The end-to-end GraphPrompterHF model shows only very little attention to the user features, but instead evens out the attention on both KGEs. We cannot tell if that means, what kind of strategy GraphPrompterHF uses to process these KGEs. The model could try to compute the similarity

between these vectors as a prediction strategy or it could extract and process the features of user and movie in another way.

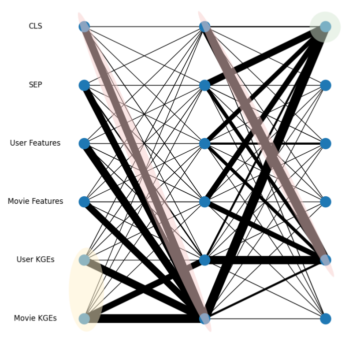
4.2.3 Attention Maps



(a) Vanilla model attention map



(b) GraphPrompterHF frozen model attention map



(c) GraphPrompterHF end-to-end model attention map

Figure 4.4: Attention maps of the three training stages

The figures 4.4 show the attention maps of each model. Figure 4.4a shows the attention map of the Vanilla model. The first layer shows no specific attentions. As we already pointed out that the residual layers transfer most information into the first layer. From first to output layer the model's classification token shows a high attentions to the separator tokens, user features and movie features. The user feature is payed the most attention to, followed by the separator tokens, followed by the movie features.
As we expected, the user features are payed the most attention to, probably because the model has to remember user preferences for each user individually, while movie feature embeddings can be generalized over the genres and other indicators.

Figure 4.4b shows the attention map of the GraphPrompterHF frozen model. The figure shows, that the movie KGEs pays a relatively high attention on all other features, specially to the classification token, user KGEs and movie KGEs, while separator tokens, user and movie features are payed less then half of the attention to. The user KGEs are paying attention on the movie KGEs in the first layer. The classification token pays high attentions to all other features in the output layer, the highest to the movie KGEs, followed by separator tokens and user features, followed by the movie features and last by far by the user KGEs. The user KGEs pays the highest attention to the classification token, followed by the user KGEs, followed by

the movie features and last by the separator tokens.
The behavior of the frozen GraphPrompterHF fits the insights we extracted from the SHAP-values. The classification token pays the most attention to the movie KGEs and the movie KGEs summarizes all other features in the first layer. The unexpected behavior of the KGEs paying high attention to the classification and separator token could be the result of the model having to figure out how to interpret the KGEs. The similarity approach by the GNN may be only interpretable by the transformer with some band-aid solution[2].

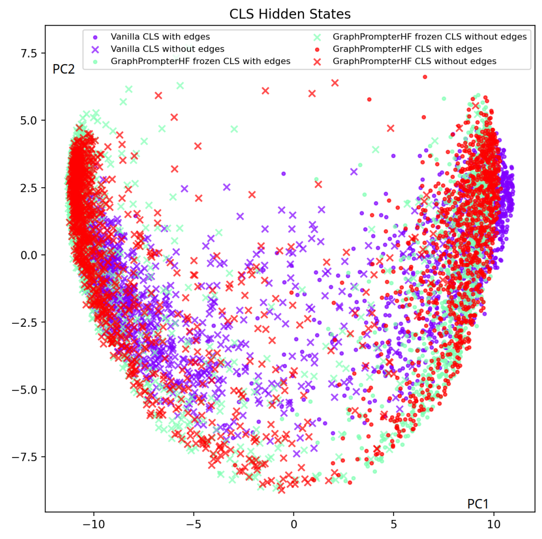
Figure 4.4c shows the attention map of the end-to-end GraphPrompterHF model. The figure shows a similar structure compared to the frozen version. This time in the first layer, each feature that was payed attention in the frozen model is now payed equally high attention by the movie KGEs. The second layers attentions also shows relative higher attentions of features that were already payed attentions to. The classification token pays higher attention to the user features and the user KGEs pays overall higher attentions to the other features that were payed attention before.
We already knew that the attention on the user features increases on this model, when looking at the SHAP-values. Overall the model does seem to use all features more equally. This could be the result of the model adjusting the GNN's parameters from the transformer standpoint. We can tell for sure, that the strategy of GraphPrompterHF end-to-end model does not change, but is strengthened instead. This could mean, that the performance loss did not stem from the model jumping out of the global minimum by following a different strategy of processing the KGEs. This only supports the theory that the performance loss is the effect of an overwrite from the previous data leak.

We marked the figures 4.4b and 4.4c with colored spheres for positions that we are most interested in analyzing more. The green circles are placed at the classifier position at the output layer. This position is most important to analyze, because it may show us the actual features the models base their prediction on. The red spheres mark the positions that are interesting to analyze that regard the influence of the classification token on the KGEs. As the classification token is a static vector, it may be a band-aid workaround of the transformer to produce static value dimensions in the embedding space, where the attention score can be mapped onto between user and movie KGEs. The yellow spheres mark the positions of the KGEs in the input layer. We are most interested in how much their quality changes before and after training them end-to-end. We can check if there is a lower or higher cosine similarity between them depending on the

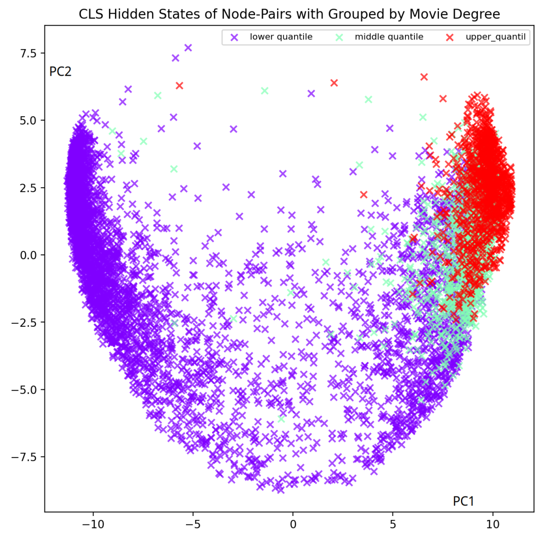
[2]Band-aid solution: a temporary solution, that does not fix the problem at it's roots.

ground-truth before and after the end-to-end training. This gives us a quantified answer to the question, whether the similarity approach by the GNN changes during the end-to-end training.

4.2.4 Hidden States of Classification Tokens



(a) Hidden states of CLS tokens grouped by models and ground truth



(b) Hidden states of CLS tokens grouped by popularity (in degree of Movie)

Figure 4.5: Hidden states of CLS tokens

Figure 4.5 shows the hidden states of CLS tokens reduced with PCA through projection on the plane spanned by the first 2 principal components on a scatter plot. Figure 4.5a shows the classification token hidden states for the Vanilla model in purple, the GraphPrompterHF frozen model in teal and the GraphPrompterHF end-to-end model in red, while the sign for hidden states that have a ground truth of *1* (has an edge between user and movie) is a circle otherwise cross. The figure shows that the hidden states ground truth separates roughly around the y-axis. The hidden states show a negative correlation between the x and the y coordinates. The closer a hidden state gets to the y-section (-8), the closer they get to the x-section (0). The cluster are shaped overall in a parabola shape.

The color distribution is more random for the Vanilla model, followed by the GraphPrompterHF end-to-end model and seems most clustered for the GraphPrompterHF frozen model. The hidden states of Vanilla model are more "out of place" for some datapoints, especially in the center of the plot.

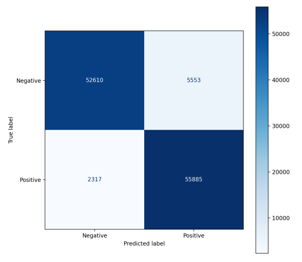
We can interpret the distribution of the classification tokens based on the model performances. Because the Vanilla model performed worst in comparison, we already expected a less clear separation of ground truths between the embeddings.
The parabola indicates that the original embeddings have a quadratic relationship between some features in their high-dimensional vector space. This form could be the result of the model separating between user-movie pairs that are hard to predict an edge for. We already assume that the movie popularity may be an important indicator for the decision process of the transformer, as some movies may be less connected and thus less represented for positive edges, but overrepresented for negative edges. If our assumption is correct, then we should be able to cluster the classification embeddings based on the degree of the movie nodes.

Figure 4.5b shows the hidden states of classification tokens grouped by the degree of edges of the movie over the entire dataset. Classification tokens whose movie degree is in the lower quantile are shown in blue. Classification tokens whose movie degree is in the middle quantile are shown in green. Classification tokens whose movie degree is in the upper quantile are shown in red. There is a clear color distribution visible. Movie embeddings in the lower quantile are dominant and take over the entire left and center field. Movie embeddings in the middle quantile are mostly positioned on the right, down to the y-axis section of -4, while the movie embeddings in the upper quantile are densed up in the top, down to the y-axis section of around -2.
We can explain some of the behaviors from our previous setup and observations. The lower representation of hidden states in the higher quantile on the left side of the plot can be explained by the fact, how we generated those false edges. We used the PyG *RandomLinkSplit* (Fey and Lenssen [2019]) to add negative train samples and this behavior does not take the original movie distribution in consideration. This resulted in movies, that are not popular to be over represented in user-movie pairs, that do not share an edge between each other.
Movies that are popular (in the middle or upper quantile) seem to be easier for the model to base their prediction on. In other words, niche movies are harder to predict for the models, while popular movies are not. The difference in distributions may have resulted in the transformer to treat unpopular movies different compared to popular movies.
The initial expectation, that the y-axis is an indicator for the models' uncertainty and the fact, that there is a correlation between movie popularity and the uncertainty factor seem to go hand-in-hand, because we can see a clear separation of colors from top to bottom, at least for positives. We can argue, that the x-axis represents the prediction output and the y-axis represents the confidence of the model. The movie popularity does not seem to be the only factor for this behavior, tho, because of the clustering behavior of negative samples (left).

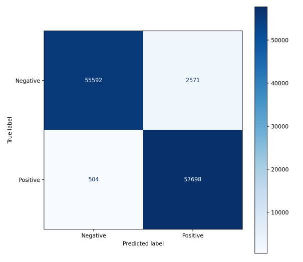
Based on our observations we can already tell, that the models focus mostly on the movies in their decision process and the more popular the movie is, the better they perform. We also noticed the over representation of unpopular movies in the negative edge data points. We should see a higher false-positive rate then false-negative rate for all models.

4.2.5 Confusion Map

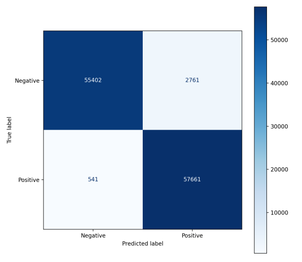
We have already explained why we expect a higher false positive rate then a false negative rate. To prove that expectation, we can compute the confusion map and with that the false positive, false negative rates and their ratio.



(a) Confusion map Vanilla model



(b) Confusion map GraphPrompterHF frozen model



(c) Confusion map GraphPrompterHF end-to-end model

Figure 4.6: Confusion maps

Figures 4.6 show the confusion maps of all models. Given the values of these confusion maps, we compute the false positive and false negative rates.

	Vanilla	**GraphPrompterHF Frozen**	**GraphPrompterHF**
False Positive Rate	0.095	0,044	0.047
False Negative Rate	0,040	0.009	0.009
Ratio	2,398	5,105	5.107

Table 4.1: False Positive and False Negative Rates in Comparison

Table 4.1 shows that the ratio between false positive rate and false negative rate is relatively high (five times as large) for the GraphPrompterHF models. The ratio for the Vanilla model is less then half.

There are more false positives then false negatives, as we expected, because the datasets provide more diverse movies for user-movie pairs that do not share an edge between each other. We can also argue, that popular movies in the dataset, where nodes share an edge between each other are over-represented and over-fit the models in that regard. One solution for that the training process could be to remove over-represented movies and or genres and balance out the dataset, so that the models have to generalize more.

4.2.6 Hidden States of KGEs

In section 4.2.2 we discussed the possibility that the GraphPrompterHF end-to-end version loses some performance, because it was overfitting on the new data points, that were not present during the GNN training. We can also assume, that the end-to-end training changes the semantic behavior of the GNN's outputs. It was trained to produce KGE with high similarity, if an edge is between two nodes. this behavior may change during the end-to-end training. If that is the case, we should see an average cosine similarity shift for user-movie nodes that share an edge between each other from high to low and a shift for nodes that do not share an edge between each other from low to high.

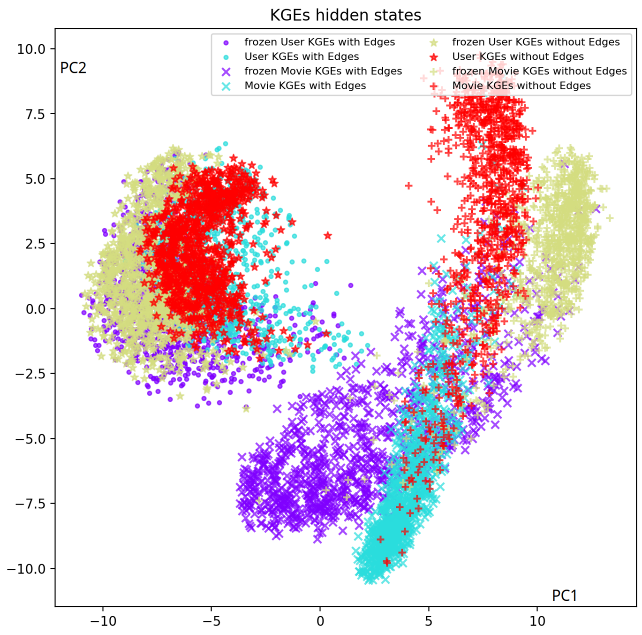


Figure 4.7: KGE hidden state distances

Figure 4.7 shows the distribution of user and movie KGEs hidden states in the first layer of the transformer in relation to each other. The plot shows the user and movie embeddings of the GraphPrompterHF frozen model with edges in purple and the embeddings without edges in yellow. The user and movie KGE hidden states of the end-ton-end version are colored in teal with edges and in red without edges. The plot shows, that user KGE hidden states are distributed close to each other to the left, while user KGE hidden states with edges are scattered a little bit more then those without edges. The movie KGE hidden states are positioned to the right and form more of a curve from top to bottom. The embeddings are less overlapping and more separated between embeddings of user-movies pairs with and without edges. The frozen movie KGE hidden states show a greater separation then the end-to-end versions. The average distance between user and movie KGE hidden states of the frozen model are on average the largest. The average distance between user and movie clusters of the frozen and end-to-end versions where there is an edge between user and movie are small and seem close to each other.
The behavior of the models are as expected. We expected the frozen version's hidden state to have a high cosine similarity if there is an edge between each other and low cosine similarity if there is no edge between each other. We also predicted that this behavior diminishes for the end-to-end model. Tho we cannot say for sure if the similarity between the frozen hidden states with edges is on average higher then the end-to-ends hidden states. The cleaner separation of the movie KGE hidden states in the frozen version were already shown in previous experiments.
To ensure that the similarity between the KGE hidden states of the frozen GraphPrompterHF are higher then the end-to-end GraphPrompterHF, we compute the average cosine similarity for all combinations and we should see the expected results.

	edge	**no edge**
frozen	0.231	-0.333
end-to-end	0.028	-0.082

Table 4.2: Cosine similarity between user and movie KGE hidden states in the first layer of frozen and end-to-end GraphPrompterHF models

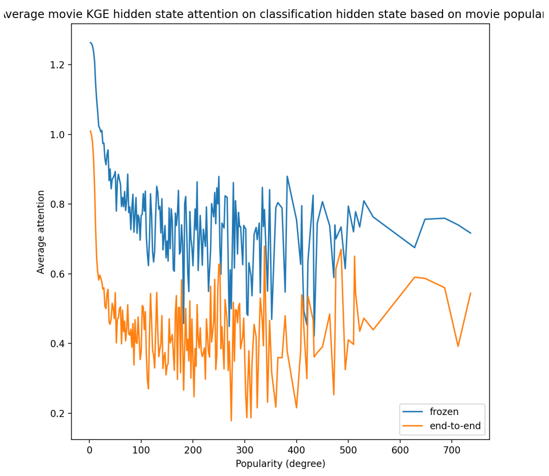
Table 4.2 shows the cosine similarities between user and movie KGE hidden states in the first layers of the GraphPrompterHF frozen and end-to-end models. The tables shows, that the similarity is ten times larger for the frozen model compared to the end-to-end model for nodes with edges between each other and almost five times smaller for frozen nodes without edges compared to end-to-end nodes without edges.

This observation indicates, that the models behave as expected and the end-to-end training diminishes the GNN's ability to produce KGEs with high and low similarity.

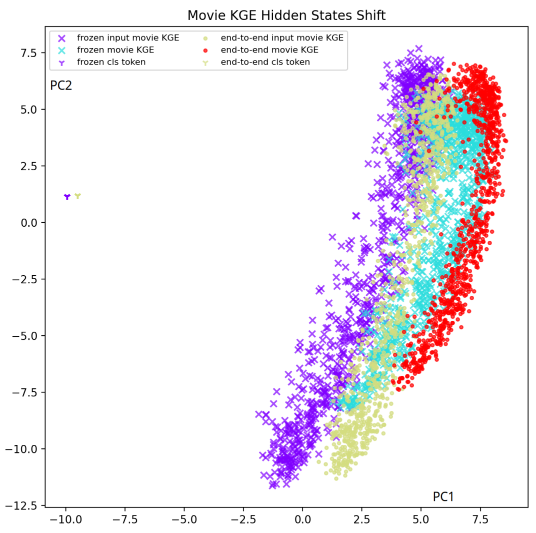
4.2.7 CLS Hidden State Influence on Movie KGE

In the last experiment we are going visualizing the influence of the classification token position on the movie KGE hidden state in the first layer. In our previous experiments we noticed a distinct output distributions for popular and unpopular movies. Popular movies tend to be classified with higher accuracy, while unpopular movies seem to confuse the model in its decision process.

The classification token hidden state is independent of the popularity of the movie. The only way this static embedding could matter (as we know from the SHAP-values) is by adjusting the attention on this static hidden state. If we see an attention difference between popular and unpopular movie KGE hidden states on the classification hidden state, then we can assume, that the models use different conceptual strategies for the prediction, one strategy that leverages the GNN and one that leverages the learned relations in the Vanilla model.



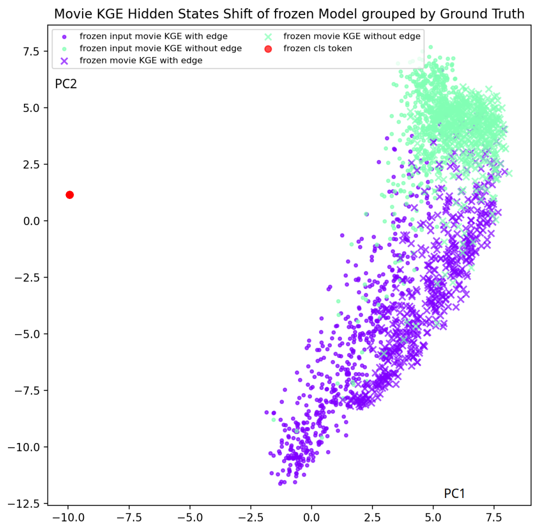
(a) Average movie KGE hidden state attention on classification hidden state



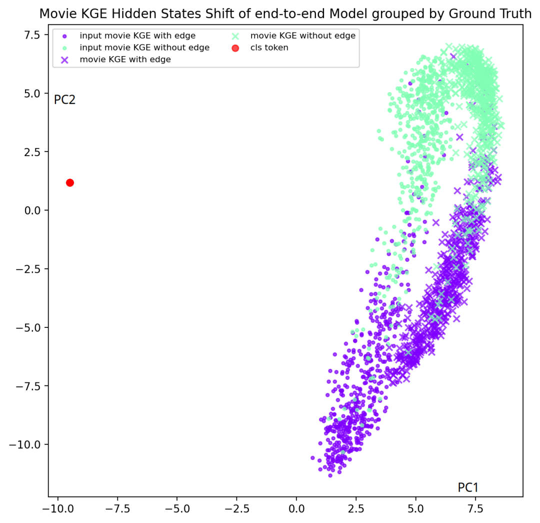
(b) Movie KGE hidden states shift by classification hidden states

Figure 4.8: Movie KGE Hidden States Shift

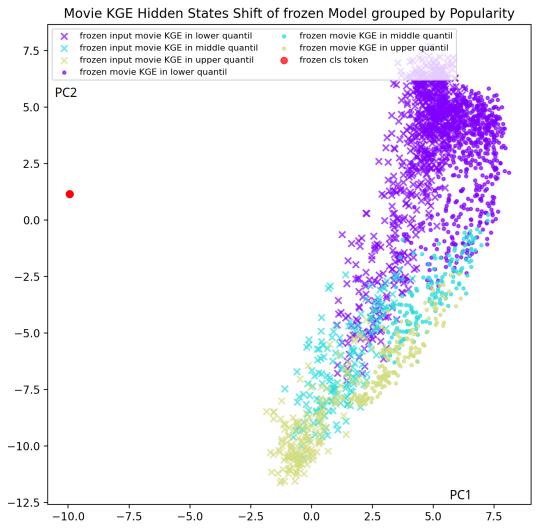
Figure 4.8b shows the movie KGE hidden state shifts from layer zero to layer one by the constants of the classification hidden states in layer zero. The classification token hidden state of the frozen GraphPrompterHF model is marked as the purple "Y" on the left, while the hidden



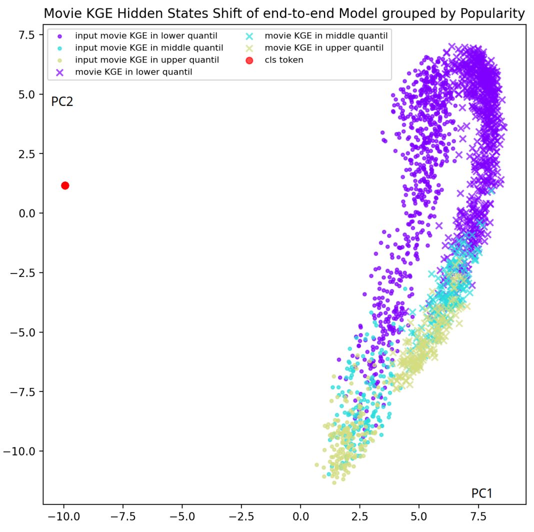
(a) Movie KGE hidden state shifts by classification hidden states of frozen model grouped by ground truth



(b) Movie KGE hidden state shifts by classification hidden states of end-to-end model grouped by ground truth



(c) Movie KGE hidden state shifts by classification hidden states of frozen model grouped by popularity



(d) Movie KGE hidden state shifts by classification hidden states of end-to-end model grouped by popularity

Figure 4.9: Movie KGE hidden state shifts

state of the end-to-end version is colored green. The movie KGE hidden states of the frozen model are marked as crosses, while the movie KGE hidden state of the end-to-end model are marked as dots. The hidden states in the initial layer are colored purple, while the hidden states in the first layer are colored teal for the frozen model. The hidden states in the initial layer are colored green and in the first layer and teal for the end-to-end model. The movie KGE hidden states all show a vertical distribution. The hidden states in the initial layers seem to be equally distributed. The hidden states in the first layer seem to be less dense around the x-axis. The horizontal lines shift to the right from the initial and to the second layer. Overall, the frozen hidden states are shifted to the left compared to the hidden states of the end-to-end model.
The classification token hidden states are very close to each other in the frozen and end-to-end phases. The frozen and end-to-end phases both seem to shift the movie KGE hidden states to the right and dense them up around the x-dimension. For the frozen version, the movie KGE hidden states seem to be also getting denser around the y-axis. For the end-to-end version, the movie KGE hidden states rather shift to the top.
These results are not telling us much about the nature of the shift. We can only tell that the shift looks similar for the frozen and end-to-end phases with the difference, that the frozen phase also increases in density on the y-dimension, while the end-to-end phase shifts more into to the top.

If we separate the views of the phases and color the hidden states by their ground truth, we should already see the difference in the shift. We expect that there will be an axis representing the ground truth of the prediction. We should see a that this dimension has a great influence on the shift.
Figure 4.9a shows the movie KGE hidden state shifts of the frozen GraphPrompterHF model from layer zero to layer one by the constant classification hidden state in layer zero grouped by ground truth. All movie KGE hidden states in the initial layer are represented by the dot while movie KGE hidden states in the first layer are represented with the cross. The hidden states who's ground truth is "there is an edge" are colored purple, while the others are colored green. The plot shows that the colors separate around the y-dimension at the height of the classification token hidden state. The lower part and largest part of this separation is colored in purple, while the upper part is colored in green. The overall distribution and shifts are similar to the distribution in the previous view.

Figure 4.9b shows the same plot for the end-to-end phase of the model. The separation between hidden states with and without edges looks overall similar. However, the separation is more of a diagonal from bottom left to top right.

We already expected one dimension to represent the ground truth of the model. In the movie KGE hidden states of the initial and first layer this dimension was the y-axis. We also expected a difference between the shift of the movie KGE embedding based on the popularity of the movie. And because we know there is a causal relation between the ground truth and the movie popularity due to the generation process of node pairs without edge in between them, we expected to see a difference in the movie KGE hidden state shift based on the ground truth.

	with edge	**without edge**
frozen	0.85	1.24
end-to-end	0.49	0.98

Table 4.3: Average movie KGE hidden state shifts

Tho it cannot be seen easily, if we calculate the average shift distance between data points before we reduced the dimension with PCA and grouped by ground truth as described in table 4.3, we can see an average shift distance difference between nodes with edge and without edge of around 150% for the frozen phase and almost 200% for the end-to-end phase. If we group the hidden states in the plots by their movie popularity, we should be able to increase this effect.

Figures 4.9c and 4.9d are equivalent plots of the movie KGE hidden state shifts grouped by movie popularity. The symbols are similar. The colors are divided into olive for movies in the lower popularity third, teal in the middle popularity third and purple in the upper popularity third.
Both plots show that movies in the upper popularity are situated at the very bottom, followed by movies in the middle third of the popularity. The movies with popularity in the lower third are placed at the top, while the borders between those groups are diagonals from bottom left to top right.
This grouping shows the effect of popularity on the shift much greater. There are great outliers of distances between cluster of movie KGE hidden states for unpopular movies (purple). Again, if we check the average attention on the hidden state based in relation to the popularity, we can expect more convincing insights.
Figure 4.8a shows the relation between the popularity of a movie and its average attention on the classification token hidden state. The figure shows two lines, while the blue line represents the frozen phase and orange the end-to-end phase of the GraphPrompterHF model. Both lines begin at a relative high average of around 1.3 for the frozen and 1.0 for the end-to-end phase. Both immediately fall steep to an average of 0.7 for the frozen phase and 0.4 for the end-to-end

phase, where they start oscillate heavily at popularity of around 100. The gap between datapoints increases and steadily.
As we suspected before, the movie popularity played a very strong role in the attention on the classification hidden state. Unpopular movies with popularity close to 1 do pay strong attention to the classification hidden state. We could argue, that the classification hidden state represents two possible strategies, that GraphPrompterHF can chose or mix in between. This behavior again suggests that the model solves the downstream task differently for popular and unpopular movies. We can also assume, that the same influence takes place to the user KGE embedding on the second layer.

4.3 Results

In this chapter we have evaluated the models performance and inner workings with explainable ai tools (XAI). In each section we have discussed certain effects of the models behavior from multiple viewpoints, and worked our way from global to local explanations. Each view on the way generated multiple following questions and with that views to produce and evaluate.
We started with the global view of model's performance in its three stages *vanilla*, *frozen* and *end-to-end*. The overall performance drop in the last phase made us beware of overfitting effects, we should avoid placing too much interpretation into. The following attention maps and SHAP-values gave us a clear indications for positions of interest, which we analyzed in detail on the internal state representations (hidden states) of the transformer.
Overall three positions of interest were analyzed in detail, namely the the classification token position in the last layer, the relationship of KGE hidden states in the input layer and the relationship of the classification token in the input layer and the movie KGE hidden states in the input and first layer.

Our experiments in section 4.2.1 showed, that the LLM's performance is improved by the usage of a the GraphPrompter architecture (Liu et al. [2024b]). The experiment also showed us the performance drop in the end-to-end phase. The performance drop drew our attention to an overfitting problem, which we complicated by a faulty transformation from the HeteroDataset (structural view of the data) into a Hugging Face dataset (NLP view of the data).
Section 4.2.2 showed the importance of the movie KGEs in the frozen phase and the alignment of other all attention scores in the end-to-end phase. In section 4.2.3 we were able to extract certain positions of interest. The first area that we became interested in were the output classification tokens. This area is the bottle neck for the downstream classification task and should carry the

most pressing features.
Our analysis of this areas' hidden states in section 4.2.4 made us aware, that one of the greatest factors in the transformer's performance is the movie's popularity. If, for example, a movie at hand is more popular, then the internal state representation of the output classification token is more reliable and spatially separated by its ground truth. If, on the other hand, a movie is less popular, then the internal state representation tends to be spread more widely and less clustered. We also recognized, that there is a great distribution difference between between user-movie-pairs, when there is actually an edge between them and pairs, when there is no edge in between them. This difference in distribution was created artificially during the generation of node pairs for which no edge exists. While node pairs for which an edge exists favor popular movies, movies in node pairs for which no edge exists are equally distributed.
In section 4.2.5 we expected the LLM to treat unpopular movies and with that node-pairs for which no edge exists differently. The confusion maps showed, that the models perform better for node-pairs for which no edge exists by a factor of two in the *vanilla* phase and five in the frozen and end-to-end phase. This models are probably biased towards the prediction of "no edge" if the movie is unpopular due the distribution differences. In other words, the model simply has to remember the popular movies and can guess that all other movies are more likely not connected with a user if the model cannot remember differently.
One other position of interest was analyzed in section 4.2.6. We analyzed how the KGE are interpreted by the LLM. By definition, KGE in the frozen phase are meant to have a high similarity between each other, if there is an edge between user and movie and low similarity, if there is no edge between them. We expected GraphPrompterHF to pick up that strategy with some band-aid mechanic, because it will have to somehow include the similarity attention score between the KGEs as an additional dimension in the actual value vector.
Our experiments showed, that the average cosine similarity between KGEs who's nodes share an edge between each other shrinks in the end-to-end phase, as well as the distance for nodes that do not share an edge between each other. This behavior indicates, that the end-to-end training diminishes the GNN's ability to produce KGE with higher and lower similarity.
The last experiment we conducted in section 4.2.7 was to analyze the influence of the static classification token on the movie KGE hidden state in the first layer. We were able to find indications, that the movie KGEs are treated differently, if the popularity of given movie is very low. This behavior becomes lower and much less predictable ones the a certain threshold of popularity was reached. We also had no reason to believe, that this behavior is different for the user KGE.

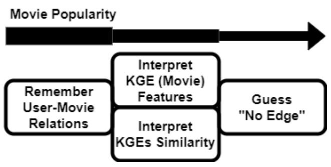

Movie Popularity
Remember User-Movie Relations
Interpret KGE (Movie) Features
Interpret KGEs Similarity
Guess "No Edge"

Figure 4.10: Strategies GraphPrompterHF follows depending on the movie popularity

Figure 4.10 illustrates the four strategies we belief GraphPrompterHF follows during a decision process. The first strategy is also followed by the Vanilla model and includes remembering specific user-movie group relationships, as we already expected in figure 4.1b. For movies of lower popularity the transformer starts to include the movie KGEs, as they can be integrated seemliness into the strategy of the Vanilla model. At the same time, the KGEs of user and movies can be compared via the band-aid attention score similarity strategy. For the least popular movies with high uncertainty, the model can chose to guess "no edge".

5 Discussion

This thesis aimed to explain the inner workings of transformers when confronted with KGEs. Using an explicit downstream task like link prediction on a combination of GRL and NLP, we explore the internal states and dynamics of the LLM and possible interpretations of these.
We were able to confirm the experiments of GraphPrompter (Liu et al. [2024b]), in which transformers benefit from soft prompting KGEs. Our experiments made us realize that in this particular setup, the models are overfitting and a bias for popular movies has developed. Furthermore, this bias and the soft prompting approach resulted in unexpected behaviors in the transformer. Especially the attention on the classification token by the KGE positions caught our attention. Our experiments gave us reason to believe that the attention on static classification tokens by the KGE positions is reflecting the transformers' attempt to interpret KGEs that have been generated under a semantically different pretext.
The GraphSage model (Hamilton et al. [2017]) we used in our experiments was trained to produce KGEs, that have high cosine similarity, if there is an edge between two given nodes and a low cosine similarity if there is no edge between them. The transformer on the other hand faces a classification problem via softmax.
Our experiments also gave us reason to believe that there is a strong relationship between the connectivity of a node and this attention effect. The attention on the static classification token was largest for nodes that have close to zero connectivity. This effect fades very quickly, which leads us to the assumption that it allows the transformer to interpret KGEs of nodes with low connectivity different then those of higher connectivity.

Our experiments underline the overall need for XAI research in fusing NLP with other ontologies and modalities. The bias of the LLM was most likely the effect of the dataset distribution, in which nodes with low connectivity are evenly distributed for node-pairs that do not share an edge with each other compared to node-pairs that do share an edge with each other. This effect is less significant in traditional GRL, as the node pairs are never analyzed in isolation, but always in a context of their neighborhood. Nodes with higher connectivity can most likely be found in the neighborhood of nodes with lower connectivity. The resulting KGEs should therefore be strongly dependent on these node centers and assigned to specific clusters. However, since the LLM only learns about this connectivity indirectly via the KGE, it can only learn this relationship by memorizing it from separate data points. The LLM resolving this issue by leveraging a band-aid strategy in which the similarity of KGEs can be interpreted directly in the attention mechanism and other strategies, that the LLM switches in between.

The effects of overfitting could have been avoided by early-stoppage and a correct transformation of the structural view to the NLP view. The effects of popular movie bias could have been avoided with multiple strategies. We could have added some of the contextual neighborhood of each node in the natural language prompt. We could have artificially generated node-pairs that do not share an edge between each other with the same distribution as there are node-pairs that do share an edge between each other. Or we could have permitted the over-representation of popular nodes to begin with. Increasing the prompt size with natural language representation of the neighborhoods could be a valid solution for the problem to some extend, but as the literature already implies, the LLM will still struggle to generalize relations between nodes as we expect them from the view of GRL. Most promising solutions would be the looking at the artificially generated biases during false-edge generation. We could balance out the dataset to begin with, so there are not overrepresented movies or we could align the amount of false-edges with the amount of true-edges for each node.

The fusion of models from different disciplines and modalities leads to unexpected behaviors. For example, the uneven distribution has led to a bias in the LLM, which did not come into play from the perspective of GRL. We can assume that other data sets, graph structures, methods, models and architectures will also lead to other unexpected behaviors. We need to analyze powerful multi-tasking language models with XAI to ensure and justify their proliferation. Specifically, we can conclude that the use of Graph-RAG should be subject to tighter analysis, especially when nodes are removed from their context and presented to language models out of context. The transfer of the KGEs used for retrieval to the LLM could be made mandatory for an interpretable and sustainable use of Graph-RAG. Fortunately, GRL and KGEs are often used as a tool for Graph-RAG and would not increase the computational effort by a lot.

Bibliography

Hasan Abu-Rasheed, Christian Weber, and Madjid Fathi. Knowledge graphs as context sources for llm-based explanations of learning recommendations. In *2024 IEEE Global Engineering Education Conference (EDUCON)*, pages 1–5, 2024. doi: 10.1109/EDUCON60312.2024.10578654.

Farahnaz Akrami, Lingbing Guo, Wei Hu, and Chengkai Li. Re-evaluating embedding-based knowledge graph completion methods. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*, CIKM '18, page 1779–1782, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450360142. doi: 10.1145/3269206.3269266.

Omar Alonso, Vasileios Kandylas, and Serge-Eric Tremblay. Scalable knowledge graph construction from twitter. *ArXiv preprint arXiv:1906.05986*, 06 2019.

Sindhu B, Prathamesh R P, Sameera M B, and KumaraSwamy S. The evolution of large language model: Models, applications and challenges. In *2024 International Conference on Current Trends in Advanced Computing (ICCTAC)*, pages 1–8, 2024. doi: 10.1109/ICCTAC61556.2024.10581180.

Jimmy Ba, Jamie Kiros, and Geoffrey Hinton. Layer normalization. *ArXiv preprint arXiv:1607.06450*, 07 2016. URL `https://www.cs.utoronto.ca/~hinton/absps/LayerNormalization.pdf`.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *ArXiv preprint arXiv:1409.0473*, 2014.

Tristan Bilot, Nour El Madhoun, Khaldoun Al Agha, and Anis Zouaoui. A survey on malware detection with graph representation learning. *ACM Comput. Surv.*, 56(11), June 2024. ISSN 0360-0300. doi: 10.1145/3664649.

Jiahang Cao, Jinyuan Fang, Zaiqiao Meng, and Shangsong Liang. Knowledge graph embedding: A survey from the perspective of representation spaces. *ACM Comput. Surv.*, 56(6), March 2024. ISSN 0360-0300. doi: 10.1145/3643806.

Long Chen, Oleg Sinavski, Jan Hünermann, Alice Karnsund, Andrew James Willmott, Danny Birch, Daniel Maund, and Jamie Shotton. Driving with llms: Fusing object-level vector modality for explainable autonomous driving. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pages 14093–14100, 2024a. doi: 10.1109/ICRA57147.2024.10611018.

Zhikai Chen, Haitao Mao, Hang Li, Wei Jin, Hongzhi Wen, Xiaochi Wei, Shuaiqiang Wang, Dawei Yin, Wenqi Fan, Hui Liu, and Jiliang Tang. Exploring the potential of large language models (llms)in learning on graphs. *SIGKDD Explor. Newsl.*, 25(2):42–61, March 2024b. ISSN 1931-0145. doi: 10.1145/3655103.3655110.

Inyoung Cheong, King Xia, K. J. Kevin Feng, Quan Ze Chen, and Amy X. Zhang. (a)i am not a lawyer, but...: Engaging legal experts towards responsible llm policies for legal advice. In *Proceedings of the 2024 ACM Conference on Fairness, Accountability, and Transparency*, FAccT '24, page 2454–2469, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400704505. doi: 10.1145/3630106.3659048.

Kyunghyun Cho, Bart van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder–decoder approaches. In Dekai Wu, Marine Carpuat, Xavier Carreras, and Eva Maria Vecchi, editors, *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*, pages 103–111, Doha, Qatar, October 2014. Association for Computational Linguistics. doi: 10.3115/v1/W14-4012.

Junyoung Chung, Çaglar Gülçehre, Kyunghyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *ArXiv preprint arXiv:1412.3555*, 2014.

Kevin Clark, Urvashi Khandelwal, Omer Levy, and Christopher D. Manning. What does BERT look at? an analysis of BERT‘s attention. In Tal Linzen, Grzegorz Chrupała, Yonatan Belinkov, and Dieuwke Hupkes, editors, *Proceedings of the 2019 ACL Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 276–286, Florence, Italy, August 2019. Association for Computational Linguistics. doi: 10.18653/v1/W19-4828.

Andy Coenen, Emily Reif, Ann Yuan, Been Kim, Adam Pearce, Fernanda Viégas, and Martin Wattenberg. *Visualizing and measuring the geometry of BERT*. Curran Associates Inc., Red Hook, NY, USA, 2019.

Zhiyong Cui, Kristian Henrickson, Ruimin Ke, and Y. Wang. Traffic graph convolutional recurrent neural network: A deep learning framework for network-scale traffic learning and forecasting. *IEEE Transactions on Intelligent Transportation Systems*, PP, 11 2019. doi: 10.1109/TITS.2019.2950416.

DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *ArXiv preprint arXiv:2501.12948*, 2025. URL `https://github.com/deepseek-ai/DeepSeek-R1`.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423.

Yuxin Dong, Shuo Wang, Hongye Zheng, Jiajing Chen, Zhenhong Zhang, and Chihang Wang. Advanced rag models with graph structures: Optimizing complex knowledge reasoning and text generation. In *2024 5th International Symposium on Computer Engineering and Intelligent Communications (ISCEIC)*, pages 626–630, 2024. doi: 10.1109/ISCEIC63613.2024.10810209.

Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. Large language models for software engineering: Survey and open problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, pages 31–53, 2023. doi: 10.1109/ICSE-FoSE59343.2023.00008.

Riya Fernando, Isabel Norton, Pranay Dogra, Rohit Sarnaik, Hasan Wazir, Zitang Ren, Niveta Sree Gunda, Anushka Mukhopadhyay, and Michael Lutz. Quantifying bias in agentic large language models: A benchmarking approach. In *2024 5th Information Communication Technologies Conference (ICTC)*, pages 349–353, 2024. doi: 10.1109/ICTC61510.2024.10601938.

Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. *ArXiv preprint arXiv:1903.02428*, 2019.

Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Qianyu Guo, Meng Wang, and Haofen Wang. Retrieval-augmented generation for large language models: A survey. *ArXiv preprint arXiv:2312.10997*, 2023.

Gemma Team and Google DeepMind. Gemma 2: Improving open language models at a practical size. *ArXiv preprint arXiv:2408.00118*, 2024.

Yoav Goldberg. Assessing bert's syntactic abilities. *ArXiv preprint arXiv:1901.05287*, 2019.

Aric Hagberg, Pieter Swart, and Daniel Chult. Exploring network structure, dynamics, and function using networkx. 06 2008. doi: 10.25080/TCWV9851.

William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 1025–1035, Red Hook, NY, USA, 2017. Curran Associates Inc. ISBN 9781510860964.

F. Maxwell Harper and Joseph A. Konstan. The movielens datasets: History and context. *ACM Trans. Interact. Intell. Syst.*, 5(4), December 2015. ISSN 2160-6455. doi: 10.1145/2827872.

Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. doi: 10.1038/s41586-020-2649-2.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016. doi: 10.1109/CVPR.2016.90.

John Hewitt and Christopher D. Manning. A structural probe for finding syntax in word representations. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4129–4138, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1419.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *ArXiv preprint arXiv:1503.02531*, 2015.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8): 1735–1780, November 1997. ISSN 0899-7667. doi: 10.1162/neco.1997.9.8.1735.

Harold Hotelling. Analysis of a complex of statistical variables into principal components. *Journal of Educational Psychology*, 24:498–520, 1933. doi: 10.2307/2278378.

J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9 (3):90–95, 2007. doi: 10.1109/MCSE.2007.55.

Filip Ilievski, Pedro Szekely, and Bin Zhang. Cskg: The commonsense knowledge graph. In *The Semantic Web: 18th International Conference, ESWC 2021, Virtual Event, June 6–10, 2021, Proceedings*, page 680–696, Berlin, Heidelberg, 2021. Springer-Verlag. ISBN 978-3-030-77384-7. doi: 10.1007/978-3-030-77385-4_41.

Ganesh Jawahar, Benoît Sagot, and Djamé Seddah. What does BERT learn about the structure of language? In Anna Korhonen, David Traum, and Lluís Màrquez, editors, *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 3651–3657, Florence, Italy, July 2019. Association for Computational Linguistics. doi: 10.18653/v1/P19-1356.

Shaoxiong Ji, Shirui Pan, Erik Cambria, Pekka Marttinen, and Philip S. Yu. A survey on knowledge graphs: Representation, acquisition, and applications. *IEEE Transactions on Neural Networks and Learning Systems*, 33(2):494–514, 2022. doi: 10.1109/TNNLS.2021.3070843.

Zhengbao Jiang, Jun Araki, Haibo Ding, and Graham Neubig. How can we know when language models know? on the calibration of language models for question answering. *Transactions of the Association for Computational Linguistics*, 9:962–977, 2021. doi: 10.1162/tacl_a_00407.

Bowen Jin, Gang Liu, Chi Han, Meng Jiang, Heng Ji, and Jiawei Han. Large language models on graphs: A comprehensive survey. *IEEE Trans. on Knowl. and Data Eng.*, 36(12):8622–8642, December 2024. ISSN 1041-4347. doi: 10.1109/TKDE.2024.3469578.

Wei Ju, Zheng Fang, Yiyang Gu, Zequn Liu, Qingqing Long, Ziyue Qiao, Yifang Qin, Jianhao Shen, Fang Sun, Zhiping Xiao, Junwei Yang, Jingyang Yuan, Yusheng Zhao, Yifan Wang, Xiao Luo, and Ming Zhang. A comprehensive survey on deep graph representation learning.

Neural Networks, 173:106207, 2024. ISSN 0893-6080. doi: https://doi.org/10.1016/j.neunet.2024.106207.

Sarvesh Prabhu Kamate, K S Venkatesh Prasad, and U N Ranjitha. Comprehending intelligent systems with explainable ai. In *2024 15th International Conference on Computing Communication and Networking Technologies (ICCCNT)*, pages 1–4, 2024. doi: 10.1109/ICCCNT61001.2024.10724937.

Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. Dense passage retrieval for open-domain question answering. In Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu, editors, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6769–6781, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.550.

Omar Khattab and Matei Zaharia. Colbert: Efficient and effective passage search via contextualized late interaction over bert. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '20, page 39–48, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450380164. doi: 10.1145/3397271.3401075.

Shima Khoshraftar and Aijun An. A survey on graph representation learning methods. *ACM Trans. Intell. Syst. Technol.*, 15(1), January 2024. ISSN 2157-6904. doi: 10.1145/3633518.

Thomas Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *ArXiv preprint arXiv:1609.02907*, 2016.

Enja Kokalj, Blaž Škrlj, Nada Lavrač, Senja Pollak, and Marko Robnik-Šikonja. BERT meets shapley: Extending SHAP explanations to transformer-based classifiers. In Hannu Toivonen and Michele Boggia, editors, *Proceedings of the EACL Hackashop on News Media Content Analysis and Automated Report Generation*, pages 16–21, Online, April 2021. Association for Computational Linguistics.

Murat Kuzlu, Zhenxin Xiao, Salih Sarp, Ferhat Ozgur Catak, Necip Gurler, and Ozgur Guler. The rise of generative artificial intelligence in healthcare. In *2023 12th Mediterranean Conference on Embedded Computing (MECO)*, pages 1–4, 2023. doi: 10.1109/MECO58584.2023.10155107.

Samuli Laato, Benedikt Morschheuser, Juho Hamari, and Jari Björne. Ai-assisted learning with chatgpt and large language models: Implications for higher education. In *2023 IEEE*

International Conference on Advanced Learning Technologies (ICALT), pages 226–230, 2023. doi: 10.1109/ICALT58122.2023.00072.

Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih, editors, *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 3045–3059, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.243.

Quentin Lhoest, Albert Villanova del Moral, Yacine Jernite, Abhishek Thakur, Patrick von Platen, Suraj Patil, Julien Chaumond, Mariama Drame, Julien Plu, Lewis Tunstall, Joe Davison, Mario Šaško, Gunjan Chhablani, Bhavitvya Malik, Simon Brandeis, Teven Le Scao, Victor Sanh, Canwen Xu, Nicolas Patry, Angelina McMillan-Major, Philipp Schmid, Sylvain Gugger, Clément Delangue, Théo Matussière, Lysandre Debut, Stas Bekman, Pierric Cistac, Thibault Goehringer, Victor Mustar, François Lagunas, Alexander Rush, and Thomas Wolf. Datasets: A community library for natural language processing. In Heike Adel and Shuming Shi, editors, *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 175–184, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-demo.21.

Youjia Li, Vishu Gupta, Muhammed Kilic, Kamal Choudhary, Daniel Wines, Wei-keng Liao, Alok Choudhary, and Ankit Agrawal. Hybrid-llm-gnn: integrating large language models and graph neural networks for enhanced materials property prediction. *Digital Discovery*, 12 2024. doi: 10.1039/D4DD00199K.

Ke Liang, Lingyuan Meng, Meng Liu, Yue Liu, Wenxuan Tu, Siwei Wang, Sihang Zhou, Xinwang Liu, Fuchun Sun, and Kunlun He. A survey of knowledge graph reasoning on graph types: Static, dynamic, and multi-modal. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 46(12):9456–9478, 2024. doi: 10.1109/TPAMI.2024.3417451.

Shuang Liang. Knowledge graph embedding based on graph neural network. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, pages 3908–3912, 2023. doi: 10.1109/ICDE55515.2023.00379.

Jerry Liu. LlamaIndex, 11 2022. URL `https://github.com/jerryjliu/llama_index`.

Yuwen Liu, Lianyong Qi, Weiming Liu, Xiaolong Xu, Xuyun Zhang, and Wanchun Dou. Graphsage-based poi recommendation via continuous-time modeling. In *Companion*

Proceedings of the ACM Web Conference 2024, WWW '24, page 585–588, New York, NY, USA, 2024a. Association for Computing Machinery. ISBN 9798400701726. doi: 10.1145/3589335.3651515.

Zheyuan Liu, Xiaoxin He, Yijun Tian, and Nitesh V. Chawla. Can we soft prompt llms for graph learning tasks? In *Companion Proceedings of the ACM Web Conference 2024*, WWW '24, page 481–484, New York, NY, USA, 2024b. Association for Computing Machinery. ISBN 9798400701726. doi: 10.1145/3589335.3651476.

Llama Team and AI @ Meta. The llama 3 herd of models. *ArXiv preprint arXiv:2407.21783*, 2024. URL `https://ai.meta.com/research/publications/the-llama-3-herd-of-models/`.

Fernando Loizides and Birgit Schmidt. Positioning and power in academic publishing: Players, agents and agendas. In *20th International Conference on Electronic Publishing*. IOS Press BV, 2016. ISBN 978-1-61499-649-1.

Scott M. Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 4768–4777, Red Hook, NY, USA, 2017. Curran Associates Inc. ISBN 9781510860964.

Wes Mc Kinney. Data sructures for statistical computing in python. In Stefan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61, 2010. doi: 10.25080/Majora-92bf1922-00a.

George A. Miller. Wordnet: a lexical database for english. *Commun. ACM*, 38(11):39–41, November 1995. ISSN 0001-0782. doi: 10.1145/219717.219748.

Ahmad Haji Mohammadkhani, Chakkrit Tantithamthavorn, and Hadi Hemmatif. Explaining transformer-based code models: What do they learn? when they do not work? In *2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 96–106, 2023. doi: 10.1109/SCAM59687.2023.00020.

Dianwen Ng, Chong Zhang, Ruixi Zhang, Yukun Ma, Fabian Ritter-Gutierrez, Trung Hieu Nguyen, Chongjia Ni, Shengkui Zhao, Eng Siong Chng, and Bin Ma. Are soft prompts good zero-shot learners for speech recognition? In *ICASSP 2024 - 2024 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 10366–10370, 2024. doi: 10.1109/ICASSP48485.2024.10447746.

Haowei Ni, Shuchen Meng, Xupeng Chen, Ziqing Zhao, Andi Chen, Panfeng Li, Shiyao Zhang, Qifu Yin, Yuanqing Wang, and Yuxi Chan. Harnessing earnings reports for stock predictions: A qlora-enhanced llm approach. In *2024 6th International Conference on Data-driven Optimization of Complex Systems (DOCS)*, pages 909–915, 2024. doi: 10.1109/DOCS63458.2024.10704454.

Zhaoyang Niu, Guoqiang Zhong, and Hui Yu. A review on the attention mechanism of deep learning. *Neurocomputing*, 452:48–62, 2021. ISSN 0925-2312. doi: https://doi.org/10.1016/j.neucom.2021.03.091.

OpenAI. Gpt-4 technical report. *ArXiv preprint arXiv:2303.08774*, 2023. URL `https://cdn.openai.com/papers/gpt-4.pdf`.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. *PyTorch: an imperative style, high-performance deep learning library*. Curran Associates Inc., Red Hook, NY, USA, 2019. URL `https://proceedings.neurips.cc/paper_files/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf`.

Karl Pearson. Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin philosophical magazine and journal of science*, 2:559–572, 1901. doi: 10.1080/14786440109462720.

Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine learning in python. *J. Mach. Learn. Res.*, 12(null): 2825–2830, November 2011. ISSN 1532-4435.

Gabrijela Perković, Antun Drobnjak, and Ivica Botički. Hallucinations in llms: Understanding and addressing challenges. In *2024 47th MIPRO ICT and Electronics Convention (MIPRO)*, pages 2084–2088, 2024. doi: 10.1109/MIPRO60963.2024.10569238.

Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. In Marilyn Walker, Heng Ji, and Amanda Stent, editors, *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies,* *Volume 1 (Long Papers)*, pages 2227–2237, New Orleans, Louisiana, June 2018. Association for Computational Linguistics. doi: 10.18653/v1/N18-1202.

Alec Radford and Karthik Narasimhan. Improving language understanding by generative pre-training. 2018. URL `https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf`.

Nils Reimers and Iryna Gurevych. Sentence-BERT: Sentence embeddings using Siamese BERT-networks. In Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan, editors, *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3982–3992, Hong Kong, China, November 2019. Association for Computational Linguistics. doi: 10.18653/v1/D19-1410.

Yinlin Ren, Shaoyong Guo, Bin Cao, and Xuesong Qiu. End-to-end network sla quality assurance for c-ran: A closed-loop management method based on digital twin network. *IEEE Transactions on Mobile Computing*, 23(5):4405–4422, 2024. doi: 10.1109/TMC.2023.3291012.

Anna Rogers, Olga Kovaleva, and Anna Rumshisky. A primer in BERTology: What we know about how BERT works. *Transactions of the Association for Computational Linguistics*, 8:842–866, 2020. doi: 10.1162/tacl_a_00349.

Sasikala. S, Arunkumar. S, Shivappriya. S. N, and Dhivyaa Sakthi. S. S. Role of explainable ai in medical diagnostics and healthcare: A pilot study on parkinson's speech detection. In *2024 10th International Conference on Control, Automation and Robotics (ICCAR)*, pages 289–294, 2024. doi: 10.1109/ICCAR61844.2024.10569414.

Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009. doi: 10.1109/TNN.2008.2005605.

Jaibir Singh, Suman Rani, and Garaga Srilakshmi. Towards explainable ai: Interpretable models for complex decision-making. In *2024 International Conference on Knowledge Engineering and Communication Systems (ICKECS)*, volume 1, pages 1–5, 2024. doi: 10.1109/ICKECS61492.2024.10616500.

Arnab Sinha, Zhihong Shen, Yang Song, Hao Ma, Darrin Eide, Bo-June (Paul) Hsu, and Kuansan Wang. An overview of microsoft academic service (mas) and applications. In *Proceedings of* *the 24th International Conference on World Wide Web*, WWW '15 Companion, page 243–246, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450334730. doi: 10.1145/2740908.2742839.

Chotanansub Sophaken, Kantapong Vongpanich, Wachirawit Intaphan, Tharathon Utasri, Chutamas Deepho, and Akkharawoot Takhom. Leveraging graph-rag for enhanced diagnostic and treatment strategies in dentistry. In *2024 8th International Conference on Information Technology (InCIT)*, pages 606–611, 2024. doi: 10.1109/InCIT63192.2024.10810521.

Timo Speith. A review of taxonomies of explainable artificial intelligence (xai) methods. In *Proceedings of the 2022 ACM Conference on Fairness, Accountability, and Transparency*, FAccT '22, page 2239–2250, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393522. doi: 10.1145/3531146.3534639.

Rui Sun, Xuezhi Cao, Yan Zhao, Junchen Wan, Kun Zhou, Fuzheng Zhang, Zhongyuan Wang, and Kai Zheng. Multi-modal knowledge graphs for recommender systems. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, CIKM '20, page 1405–1414, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450368599. doi: 10.1145/3340531.3411947.

Jiabin Tang, Yuhao Yang, Wei Wei, Lei Shi, Lixin Su, Suqi Cheng, Dawei Yin, and Chao Huang. Graphgpt: Graph instruction tuning for large language models. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '24, page 491–500, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400704314. doi: 10.1145/3626772.3657775.

Jie Tang, Jing Zhang, Limin Yao, Juanzi Li, Li Zhang, and Zhong Su. Arnetminer: extraction and mining of academic social networks. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '08, page 990–998, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605581934. doi: 10.1145/1401890.1402008.

Wilson L. Taylor. "cloze procedure": A new tool for measuring readability. *Journalism Quarterly*, 30(4):415–433, 1953. doi: 10.1177/107769905303000401.

Ian Tenney, Dipanjan Das, and Ellie Pavlick. Bert rediscovers the classical nlp pipeline. In *Annual Meeting of the Association for Computational Linguistics*, 2019.

Marina Tropmann-Frick, Hannu Jaakkola, Bernhard Thalheim, Yasushi Kiyoki, and Naofumi Yoshida, editors. *Information Modelling and Knowledge Bases XXXV*, volume 380 of *Frontiers* *in Artificial Intelligence and Applications*. IOS Press, Netherlands, 2024. ISBN 978-1-64368-476-5. International Conference on Information Modeling and Knowledge Bases ; Conference date: 05-06-2023 Through 09-06-2023.

Iulia Turc, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Well-read students learn better: On the importance of pre-training compact models. *ArXiv preprint arXiv:1908.08962*, 2019.

Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009. ISBN 1441412697.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6000–6010, Red Hook, NY, USA, 2017. Curran Associates Inc. ISBN 9781510860964.

Jesse Vig. A multiscale visualization of attention in the transformer model. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 37–42, Florence, Italy, July 2019. Association for Computational Linguistics. doi: 10.18653/v1/P19-3007.

Egor N. Volkov and Alexey N. Averkin. Local explanations for large language models: a brief review of methods. In *2024 XXVII International Conference on Soft Computing and Measurements (SCM)*, pages 189–192, 2024. doi: 10.1109/SCM62608.2024.10554222.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. Transformers: State-of-the-art natural language processing. In Qun Liu and David Schlangen, editors, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-demos.6.

Lingfei Wu, Yu Chen, Kai Shen, Xiaojie Guo, Hanning Gao, Shucheng Li, Jian Pei, and Bo Long. *Graph Neural Networks for Natural Language Processing: A Survey*. 01 2023. ISBN 978-1-63828-142-9. doi: 10.1561/9781638281436.

Yonghui Wu, Mike Schuster, Z. Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin

Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason R. Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Gregory S. Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. *ArXiv preprint arXiv:1609.08144*, 2016.

Zhenqin Wu, Bharath Ramsundar, Evan N. Feinberg, Joseph Gomes, Caleb Geniesse, Aneesh S. Pappu, Karl Leswing, and Vijay Pande. Moleculenet: a benchmark for molecular machine learning. *Chem. Sci.*, 9:513–530, 2018. doi: 10.1039/C7SC02664A.

Min Xu and Yongchao Yin. A similarity index algorithm for link prediction. In *2017 12th International Conference on Intelligent Systems and Knowledge Engineering (ISKE)*, pages 1–6, 2017. doi: 10.1109/ISKE.2017.8258724.

Ziqi Yang, Xuhai Xu, Bingsheng Yao, Ethan Rogers, Shao Zhang, Stephen Intille, Nawar Shara, Guodong Gordon Gao, and Dakuo Wang. Talk2care: An llm-based voice assistant for communication between healthcare providers and older adults. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 8(2), May 2024. doi: 10.1145/3659625.

Yi Yu, Ali Tosyali, Jaeseung Baek, and Myong K. Jeong. A novel similarity-based link prediction approach for transaction networks. *IEEE Transactions on Engineering Management*, 71: 981–992, 2024. doi: 10.1109/TEM.2022.3146037.

Seongjun Yun, Minbyul Jeong, Raehyun Kim, Jaewoo Kang, and Hyunwoo J. Kim. Graph transformer networks. *ArXiv preprint arXiv:1911.06455*, 2020.

Jiachi Zhang, Wenchao Zhou, and Benjamin E. Ujcich. Provenance-enabled explainable ai. *Proc. ACM Manag. Data*, 2(6), December 2024. doi: 10.1145/3698826.

Haiyan Zhao, Hanjie Chen, Fan Yang, Ninghao Liu, Huiqi Deng, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, and Mengnan Du. Explainability for large language models: A survey. *ACM Trans. Intell. Syst. Technol.*, 15(2), February 2024. ISSN 2157-6904. doi: 10.1145/3639372.

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 13. März 2025 Ahmad Khalidi