University of Stuttgart

Institute for Artificial Intelligence
Analytic Computing

Universitätsstraße 32
70569 Stuttgart

Master Thesis

Byte-Pair-Encoding for Knowledge Graph Embeddings

Fatos Ferati

Study program:	M.Sc. Informatik
1. Examiner:	Prof. Dr. Steffen Staab
2. Examiner:	Prof. Dr. Mathias Niepert
Advisors:	Cosimo Gregucci
start date:	27.01.2025
end date:	27.07.2025

Statement

I hereby certify that this thesis has been composed by me and is based on my own work, that I did not use any further resources than specified – in particular no references unmentioned in the reference section – and that I did not submit this thesis to another examination before. The paper submission is identical to the submitted electronic version.

	Yes	No
I agree to have this thesis published in the library.	☒	☐
I agree to have this thesis published on the Web.	☒	☐
The thesis text is available under a Creative Commons License (CC BY-SA 4.0).	☒	☐
The source code is available under a GNU General Public License (GPLv3).	☒	☐
The collected data is available under a Creative Commons License (CC BY-SA 4.0).	☒	☐

....

(Place, Date) (Signature)

Note

- If you would like us to contact you for the graduation ceremony,
  please provide your personal E-mail address: fatos_fe@hotmail.de
- If you would like us to send you an invite to join the AC Alumni
  and Members group on LinkedIn, please provide your LinkedIn ID : Fatos-Ferati

Zusammenfassung

Wissensgraphen (Knowledge Graphs, KGs) sind ein zentrales Mittel zur strukturierten Repräsentation großer Informationsmengen. Sie sind jedoch von Natur aus unvollständig, da Verbindungen jederzeit fehlen können. Deswegen ist die Vorhersage fehlender Verbindungen – bekannt als *Link-Prediction* – eine zentrale Aufgabe. Link-Prediction wird in verschiedenen Kontexten durchgeführt, nämlich *transduktiv*, wo alle Entitäten und Beziehungstypen während des Trainings beobachtet werden; *induktiv*, wo einige Entitäten nicht sichtbar sind; und *vollständig induktiv*, wo einige Entitäten und Beziehungstypen nicht sichtbar sind. Es wurden erhebliche Anstrengungen unternommen, um Link-Prediction im transduktiven Umfeld zu verbessern, wobei sich Knowledge Graph Embedding (KGE) als ein prominenter Ansatz herauskristallisiert hat. Während transduktive Methoden gut erforscht sind, benötigen bestehende Ansätze für induktive Szenarien oft zusätzliche Informationen über die Trainingsdaten hinaus. Um den vielerforschten Bereich der transduktiven Link-Prediction auszunutzen, wurde kürzlich BYTE als Ansatz vorgeschlagen. Dieser passt transduktive Embedding-Modelle mithilfe eines BPE-Tokenizers an induktive Aufgaben an. Allerdings basiert diese Methode auf einem vordefinierten Token-Vokabular und wurde nur auf Benchmarks mit wenigen Verbindungen mit unbekannten Entitäten evaluiert. Diese Arbeit baut auf BYTE auf, indem sie speziell entwickelte Token-Vokabulare einführt, den Prozess der Umwandlung von Token-Embeddings in Entitäts- und Beziehungs-Embeddings modifiziert und anschließend eine systematische Bewertung anhand von Standard-Induktionsbenchmarks durchführt.

Abstract

In the field of knowledge representation, Knowledge Graphs (KGs) play a crucial role in organizing and representing vast amounts of information. However, they are inherently incomplete, as links can be missing from the graph. The task of predicting such missing links is known as *link prediction*. Link prediction is performed in different settings, namely *transductive*, where all entities and relation types are observed during training; *inductive*, where some entities are unseen; and *fully-inductive*, where some entities and relation types are unseen. A considerable amount of effort has been devoted to improving link prediction in the transductive setting, with Knowledge Graph Embedding (KGE) emerging as a prominent approach. While research on inductive and fully-inductive link prediction has been conducted, existing methods rely on additional information beyond the set of training edges. To leverage the rich transductive literature for the two inductive settings, recent work in the form of BYTE proposed using a BPE tokenizer to adapt transductive KGE to the inductive settings. However, it relies on a pre-constructed token vocabulary that may not be suitable for all knowledge graphs, and its evaluation is limited to benchmarks that

contain only a few test edges with unseen elements. This thesis builds upon ByTE by introducing purpose-built token vocabularies, modifying the process of transforming token embeddings into entity and relation embeddings, and subsequently performing a systematic evaluation on standard inductive benchmarks.

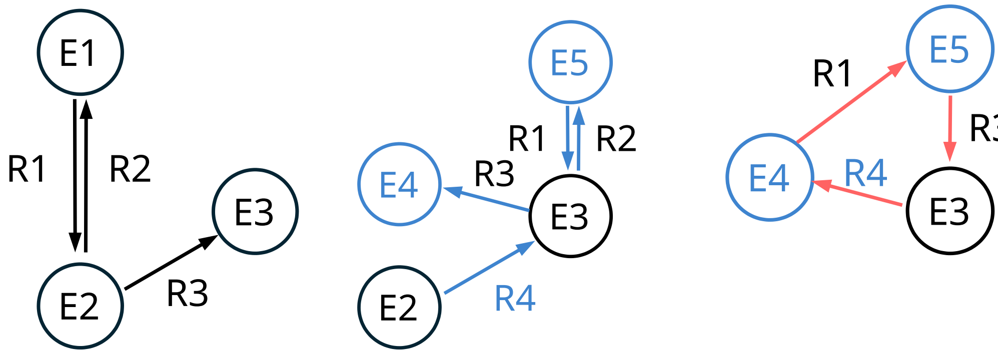
Contents

1. Introduction

In the field of knowledge representation, Knowledge Graphs (KGs) play a crucial role in organizing and representing vast amounts of information. Specifically, a knowledge graph consists of triples, representing relationships between entities. However, despite their increasing size, KGs are usually incomplete, often missing edges or entities [1]. One of the most prominent tasks for addressing this incompleteness is predicting missing edges, commonly referred to as link prediction. One major approach for link prediction is Knowledge Graph Embedding (KGE). KGE aims to find a mapping function that transforms plain data in the form of entities and relation types into a dense, low-dimensional space suited for link prediction [2]. Both the training of the mapping functions and the subsequent inference of new links are done using scoring functions. These functions receive the embeddings for the entities and relation types involved in a link and output the score of the link. One common assumption in KGEs is the knowledge of all possible entities and relation types (transductive setting); however, KGEs can also be used in settings where previously unseen entities (inductive setting) or both unseen entities and relation types (fully-inductive setting) can appear [3, 4]. Although many methods [3–9] have been proposed to perform link prediction in both inductive and fully-inductive settings, they suffer from additional drawbacks compared to traditional KGE methods. Some are computationally expensive [7, 8], while others cannot perform link prediction on unseen entities without additional relational context [3, 4, 9], and still others require data beyond the set of edges [5, 6].

However, a recent development is the tokenization-based approach BYTE [10]. BYTE is a generic approach that allows any traditional (transductive) KGE method to be used in an inductive setting. For this, the embedding of an entity or relation type is constructed from its tokenization generated from a token vocabulary. As a result, the requirement of static entity and relation type sets is relaxed to a static set of tokens. In other words, any KG element can be embedded as long as it can be tokenized. In order to ensure that this requirement is met, BYTE makes use of a prebuilt BPE token vocabulary [11]. While similar approaches exist, BYTE is unique in requiring minimal additional information in the form of linguistic labels of entity and relation types. This approach has shown promise in demonstrating a potential for fully-inductive link prediction. However, the evaluation in their work focused on hand-crafted examples and benchmarks in which only a small proportion of edges included unknown elements. In particular, their work lacks experiments on benchmarks using established *inductive* datasets. Therefore, it is unclear how BYTE compares with the state-of-the-art inductive methods.

Building upon BYTE [10], the scope of this work is threefold. First, BYTE uses a token vocabulary built from datasets used for training large language models. Therefore, the vocabulary contains tokens irrelevant to the graph, while potentially discarding tokens that are relevant to it. We believe that a custom token vocabulary built from the KG before the actual training of BYTE will be more effective for



(a) Training graph. (b) Inference graph. (c) Test graph.

Figure 1: Visualization of knowledge graph partitions used in link prediction. Graph elements are represented by circles (entities), arrows (edges), and edge labels (relation types). Blue elements are first observed during inference, while red elements remain completely unobserved.

link prediction. Second, they only demonstrate BYTE's capability of outperforming the transductive counterpart of the KGE methods; neither results on inductive benchmarks nor comparisons to state-of-the-art inductive methods were shown. We, therefore, systematically evaluated BYTE on multiple commonly used datasets. Third, we identified and evaluated multiple improvements to the original BYTE approach on the same datasets.

The rest of this work is structured as follows. In Section 2, we provide the required background for this work. In Section 3, we give an overview of the related work. In Section 4, we formalize our research questions. In Section 5, we present our approach for creating custom vocabularies and the modifications to the BYTE approach. In Section 6, we show our evaluation methodology and the results of our work. In Section 7, we summarize our results and discuss possible future work.

2. Background

This section is structured as follows. In Section 2.1, we introduce knowledge graphs and link prediction. In Section 2.2, we explain the different types of tokenization and the BPE tokenizer, which is a core component of the BYTE approach. In Section 2.3, we explain how training on sequential data is handled.

2.1. Knowledge Graphs and Link Prediction

A **Knowledge Graph (KG)** is a directed, labeled graph (V, R, E), where V represents the set of entities, R the set of relation types, and $E \subset V \times R \times V$ the set of triples. Finally, graphs are often stored as triples (h, r, t), where h and t are the head and tail entities, and r is the relation type linking h and t. For the majority of KGE methods, information on entities and relation types is limited to the relational information contained in E. Any additional information, such as titles or descriptions, is reduced to arbitrary indices. For this work, however, we instead assign each element of the graph a linguistic label similar to those found in popular standards for storing triple-based KGs, such as RDF [12]. Consequently, we use the linguistic label and the corresponding entity or relation interchangeably.

One of the most common tasks in the context of KGs is **link prediction**. Given two disjoint subgraphs of a KG G, namely the *train graph* $G_{\text{train}} = (V_{\text{train}}, R_{\text{train}}, E_{\text{train}})$, and the *test graph* $G_{\text{test}} = (V_{\text{test}}, R_{\text{test}}, E_{\text{test}})$, such that $G = G_{\text{train}} \cup G_{\text{test}}$, the goal of link prediction is to train a prediction model on G_{train} and evaluate it on G_{test}. This model is used to predict whether a triple (h, r, t) is part of G. This prediction should be accurate for any edge in E_{test}. Additionally, it is common to withhold a set of edges E_{valid} from E_{test}. As $E_{\text{valid}} \cap E_{\text{test}} = \emptyset$, validation edges can be used to guide hyperparameter tuning and early stopping without exposing the model to the actual test edges. This ensures better generalization and unbiased evaluation.

Link prediction can be of three types depending on the set of entities and relation types:

1. **Transductive**: $V_{\text{train}} = V$, $R_{\text{train}} = R$. There are neither unseen entities nor relation types in G_{test}.

2. **Inductive**: $V_{\text{train}} \subsetneq V$, $R_{\text{train}} = R$. There are unseen entities but no unseen relation types in G_{test}.

3. **Fully-inductive**: $V_{\text{train}} \subsetneq V$, $R_{\text{train}} \subsetneq R$. There are both unseen entities and unseen relation types in G_{test}.

In inductive settings, it is common [3, 4] to split the original graph G into three disjoint subsets for training, inference, and testing, such that $G = G_{\text{train}} \cup G_{\text{inf}} \cup G_{\text{test}}$ (see Fig. 1). Generally, all unseen test entities and relation types are present in the inference graph; thus, $V_{\text{test}} \subset V_{\text{train}} \cup V_{\text{inf}}$ and $R_{\text{test}} \subset R_{\text{train}} \cup R_{\text{inf}}$ hold. Training is conducted exclusively on G_{train}, where the model learns to predict links using entities and relation types seen in that subset. Instead of relying on fixed embeddings, the model learns transferable patterns that generalize to unseen entities and relation types in G_{inf} and G_{test}. Notably, G_{inf} is only used to detect previously learned patterns in unseen entities and relation types; the model parameters remain fixed during inference.

2.2. Tokenization

A standard approach to language processing is to take linguistic data sets, called corpora, and split them up according to a static set of inputs. This set of inputs, called the token vocabulary, is then used to train models. The process of creating such vocabularies is referred to as *tokenization*. The primary challenge lies in selecting appropriate tokens while maintaining a manageable vocabulary size. On one hand, a more extensive vocabulary would allow for a greater diversity of tokens, enabling more nuanced analysis. However, it also leads to sparser data, with many tokens occurring infrequently, which can hinder the accuracy of model training. On the other hand, a smaller vocabulary would contain fewer tokens that appear too rarely for accurate model training. The result is a more generalizable vocabulary. [13, 14]

Based on these principles, three approaches to tokenization can be identified:

Word tokenization: Tokens are chosen based on static rules. This includes using set delimiters (Whitespace Tokenizer) or regular expressions (Treebank Word Tokenizer). This approach results in larger vocabularies that cannot generalize to unseen words. [13]

Character tokenization: Tokens are constrained to a single character. This approach puts less emphasis on the meaning of entire words. However, it is less affected by typos and noisy data and is particularly suited for logographic languages, such as Chinese. [13]

Subword tokenization: Tokens are chosen as a middle ground between character and word tokenization. Subword tokens are more distinct than character tokens and allow the reconstruction of all words using a chosen alphabet. Such tokenizers include BPE [15, 16], WordPiece [17], and Unigram Language Model [18].

For this work, **Byte Pair Encoding (BPE)** [15, 16] plays a significant role. In BPE, an initial vocabulary, such as the alphabet, is used to ensure that any string in the corpus can be written as a sequence of tokens. In each iteration, the algorithm scans the entire corpus for adjacent token pairs, counting each occurrence. Among these, only the most frequent pair is selected for merging and subsequently added to the vocabulary. Since no tokens are removed, the new vocabulary can reconstruct any string that the initial vocabulary can. The result after a predetermined number of steps is the final token vocabulary, which is fitted to a given corpus. This vocabulary represents words in similar corpora using fewer tokens, while focusing on subwords that are shared across many strings. Additionally, BPE prioritizes common tokens and, therefore, generalizability, while also limiting the vocabulary size. Of particular relevance to this work is *byte-level BPE*. It uses the set of bytes as its initial vocabulary and is known for its applicability to any type of corpus, as well as its use in GPT-2 [11]. BPE has three main advantages: (1) BPE does not require any assumptions about the content of the corpus; (2) It allows for the creation of vocabularies of a deliberately chosen size; (3) Its subword tokens can be used to analyze unknown words and identify shared components.

2.3. Machine learning and sequential inputs

Embeddings are a foundational component of many machine learning approaches. They transform raw input data into fixed-size numerical vectors (or matrices) that can be processed efficiently by algorithms. These embeddings capture important features and relationships within the data, enabling models to understand and work with complex inputs, such as text, images, or speech, more effectively.

For fixed-size inputs, the data can be mapped directly into the embedding space through a projection. For example, each word in a vocabulary can be associated with a corresponding embedding vector. This projection is typically implemented as a simple lookup table that retrieves the embedding for each input token.

However, for variable-length—and particularly sequential—inputs, such as video or speech, this approach is no longer sufficient. Several methods have been proposed to address this challenge. These include transforming the input into a fixed-size representation before embedding it; employing various types of neural networks (e.g., RNNs [19], Transformers [20]) to generate embeddings directly from the input sequence; and applying aggregation techniques to combine individual embeddings into a fixed-size vector.

In the scope of this work, we handle inputs in the form of variable-sized sequences of fixed-size embeddings. For this, we explored two approaches. The first approach, as employed in BYTE, involves **normalizing input length**. A fixed input size is predefined, and all inputs are adjusted accordingly: inputs shorter than this length are padded, while longer inputs are truncated to fit the specified size. Alternatively, tokens may be removed according to an ordering strategy that differs from truncation.

The second approach involves the use of **Recurrent Neural Networks (RNNs)**. Unlike simple NNs, RNNs are well-suited for processing sequential inputs of variable length. They maintain a hidden state that is updated at each step as elements of the input sequence are processed in order. The final hidden state can then be used as an embedding of the entire input sequence. RNNs are widely used for modeling sequential data, including real-world applications related to linguistics. For this work, three types of RNNs are of importance: (Simple) RNNs [21], Gated Recurrent Units (GRUs) [22], and Long Short-Term Memory (LSTM) networks [23]. Simple RNNs struggle to retain information from earlier parts of the sequence, which can make them less suitable for capturing long-range dependencies. GRUs address this limitation by introducing two gating mechanisms that help regulate the flow of information over time, improving long-range memory. LSTM networks are the precursor of GRUs, with three gates and an additional memory cell. They are more complex than GRUs, but achieve even better results on long sequences.

3. Related work

This section is structured as follows. In Section 3.1, we discuss KGE methods in a transductive context. In Section 3.2, we discuss different types of methods used in link prediction. In Section 3.3, we explain the mechanisms behind BYTE.

3.1. KGE in the transductive setting

While there are various approaches to transductive link prediction, this work examines embedding-based models. In such methods, a function f is learned to map entities and relation types to a d-dimensional embedding space $\mathbb{F}^d$ over a field $\mathbb{F}$. These embeddings are then passed into score functions to estimate the likelihood of an edge being true.

The majority of methods differ primarily in the choice of the field underlying the embedding space and the scoring functions. For example, translational methods such as TransE [24] score the embeddings by measuring the distance between the embedding of the tail entity and the sum of the embeddings of the head entity and the relation type. Other methods, such as DistMult [25], instead use linear operations to compute the scores. In terms of embedding space, most of them are based on the field of real numbers $\mathbb{R}$, while some methods, such as ComplEx [26], use spaces based on other fields, such as the field of complex numbers $\mathbb{C}$. However, most methods in this category share the same drawbacks [10], namely, (1) the size of the embedding layer increases linearly with the number of entities, and (2) unseen entities and relation types cannot be recognized and thus embedded.

In this work, we will utilize two transductive KGE methods. The first is the previously mentioned ComplEx [26]. The second method is Keci [27]. Keci uses Clifford Algebras $Cl_{p,q}$ as the basis of their embedding space. Clifford Algebras allow for the generalization of many fields used in embedding spaces, such as $\mathbb{R} \subseteq Cl_{0,0}$. The selection of the underlying field is therefore a part of the training. These two methods were chosen because they are well-regarded, have competitive performance, and are also used in BYTE.

3.2. Link prediction in the inductive setting

During our research on inductive link prediction, we have identified different approaches. Unlike the previously stated methods utilizing KGE (see Section 3.1), these approaches are based on learning attributes of the graph that extend beyond the set of edges. As a result, the following approaches are all applicable to at least the inductive setting.

Some works, such as GraphSAGE [5] and BLP [6], perform inductive link prediction by utilizing additional node or relation type information. In GraphSAGE, embeddings of entities are computed by aggregating the features of their neighbors and combining them with the entities' own features. BLP instead uses textual descriptions to embed entities. Because this approach utilizes features instead of entities to generate embeddings, it is suitable for inductive settings. It can also be extended to allow embedding relation types, as is the case for RAILD [28]. A recent method following this approach is BYTE [10]. Similar to BLP, this method utilizes additional data in the form of textual information. However, the data consists solely of the textual labels of entities and relation types.

In another approach, DRUM [7] and AnyBURL [29] rely on mining rules. In these methods, entities and relation types are not embedded. Instead, rules are mined during training and then evaluated during inference. Both methods reason over relation types and can therefore only handle unseen entities.

Finally, there are methods that reason over graph structures. This includes methods such as GraIL [3] and CSR [8], which utilize the local context of entities captured through subgraphs. Additionally, CSR utilizes meta-learning to generalize seen relations to unseen ones, enabling fully-inductive link prediction. Other methods, such as NodePiece [9], and ULTRA [4], instead gather data directly from the surroundings of elements of the graph. In addition, ULTRA stands out from the named methods by transforming graphs into graphs of relations, which encode the relative connections between relation types. The resulting graph edges are then generalized across all possible entity graphs, enabling fully-inductive reasoning. This approach has the drawback that predicting links with unseen elements can only be done if substructures containing those elements are available during inference (see figure Fig. 1). For example, NodePiece encodes entities with outgoing edges and nearby entities as input, creating an embedding from the resulting encoding. Predicting an unseen entity without known links will result in poor link prediction performance.

3.3. The Byte-Pair Encoding layer: BYTE

BYTE is an approach based on adding a layer before transductive KGE methods. Similar to the previously mentioned BLP, this layer utilizes additional data in the form of textual information. However, the data consists solely of the textual labels of entities and relation types. These labels are far more commonly available than other forms of node or relation type information. The output of this layer is an embedding that can be processed by transductive KGE methods such as ComplEx or Keci.

The BYTE approach consists of using m-sized tokenizations of entities and relation types ($tok : V \times R \times V \mapsto \mathbb{N}^m \times \mathbb{N}^m \times \mathbb{N}^m$) created by a pre-built BPE-tokenizer vocabulary used in GPT-2 [11], followed by embedding the tokens ($emb : \mathbb{N}^m \times \mathbb{N}^m \times \mathbb{N}^m \mapsto \mathbb{R}^{m \times d} \times \mathbb{R}^{m \times d} \times \mathbb{R}^{m \times d}$). The matrix of token embeddings is then reduced by mapping it to the embedding space of a given transductive KGE method ($Lin : \mathbb{R}^{m \times d} \times \mathbb{R}^{m \times d} \times \mathbb{R}^{m \times d} \mapsto \mathbb{R}^d \times \mathbb{R}^d \times \mathbb{R}^d$) via a linear function. The benefit of utilizing the BPE-tokenizer is that it ensures any entity or relation type can be tokenized and thus embedded.

Table 1 demonstrates how tokenizing entities works in BYTE. Each entity and relation type is first transformed into m-sized arrays of tokens. Array sizes are ensured through padding and truncation, as seen with "england" and "new england" in the example. These arrays are then embedded via a lookup table, resulting in one d-sized embedding for each token. These token embeddings differ based on both the token and its position. Therefore, the embedding for "eng" will be the same for "england" and "english", but different for "new england". The token embeddings for each array are then flattened and embedded again. The final embedding of an entity is then a single d-sized vector encoded in the space of a chosen transductive KGE model.

Table 1: BYTE tokenization mock example with $m = 3$, $d = 1$

Entity	england			english			new england		
Tokenization	eng	land	\t	eng	l	ish	new	' '	eng
Token embedding	1.3	3.7	0	1.3	0.2	2.1	2.5	1.7	3.1
Entity embedding	3.5			1.6			4.2		

This approach has multiple advantages. One advantage is the low requirement for the training data. As long as entities and relation types are stored in a linguistically meaningful form, embeddings can be generated. Furthermore, unlike rule-mining-based methods such as DRUM, BYTE's parameter count does not grow exponentially with the number of entities and relation types. Lastly, unlike substructure-based methods such as ULTRA, BYTE can perform link prediction without relying on additional relational information in the form of an inference graph during inference.

However, we see four possible shortcomings to BYTE. First, while BYTE has shown improvements in link prediction compared to its transductive counterparts, matrices for both the token embeddings and the final embeddings are learned, increasing the number of features. As a result, it is unclear whether the improvements in transductive link prediction performance are due to a more complex model or the use of the BPE tokenizer. Second, BYTE does not have a mechanism to explore smaller tokens contained in seen tokens. Therefore, many unmerged tokens remain unseen, staying in their initialized state. In addition to all the tokens that are not found in the dataset at all, this results in parameter bloat and poor performance on sufficiently different labels. Third, as mentioned above, the addition or deletion of tokens is necessary for the token arrays to be of equal size. Using the same example in Table 1, simple truncation caused the tokenizations of two otherwise similar entities ("new england", "england") to diverge ("new eng", "england"), dropping the meaningful token "land". This might negatively affect results for KGs containing entities with a highly variable number of subword tokens. Fourth, the token vocabulary used is built for large language models. Therefore, the tokenizer is aware of tokens outside the graph, which introduces biases while also discarding tokens that should have been in the vocabulary based on the graph.

4. Research questions

Our work examines the feasibility of traditional transductive KGE methods in a (fully-)inductive setting using textual labels of entities and relation types.

For this, we build on top of the recently introduced approach BYTE [10], aiming to answer the following research questions:

RQ1: How does BYTE compare to state-of-the-art methods with regard to link prediction in inductive and fully-inductive settings?

RQ2: How can we build custom token vocabularies specialized for link prediction with transductive KGE?

RQ3: Does using a custom token vocabulary improve the inductive and fully-inductive link prediction performance of BYTE?

RQ4: How can we improve the process of transforming token embeddings to label embeddings?

We address RQ2 in Section 5.2 and our chosen approaches to RQ4 in Section 5.3, Section 5.4, and Section 5.5. RQ1 and RQ3 are addressed in Section 6.

5. Methodology

This section presents the core methodological contributions of our work. We propose a series of methods designed to enhance the representation of entities and relations in knowledge graphs through token-based embeddings. These include the construction of token vocabularies derived from knowledge graphs, preprocessing tokenizations for embedding, and an improved strategy for combining token embeddings into label embeddings.

The remainder of this section is organized as follows. In Section 5.1, we discuss the requirements that an embedding layer like BYTE should fulfill. In Section 5.2, we describe our implementation of a KG tokenizer based on BPE. In Section 5.3, we describe our first modification, which concerns the truncation of tokenizations. In Section 5.4, we describe how the linear function can be replaced through the use of RNNs. In Section 5.5, we describe how we can increase the number of tokens that are explored.

5.1. Requirements

We set three requirements for an embedding layer:

1. **Applicability:** Any entity or relation must be embeddable, and therefore tokenizable. This requirement must be considered during vocabulary construction to ensure that the resulting vocabulary can represent all elements in the KG.

2. **KG-tailored:** The tokenization of labels should focus on subword tokens that are important for labels common in KGs. This requirement guides the creation of custom vocabularies.

3. **Faithfulness:** Tokenizations and embeddings should remain faithful to the entity or relation. Specifically, they should preserve key properties of the input and differences between inputs. For example, the tokenizations of the two entities "American English" (13,27) and "British English" (19,27) would be considered faithful, while (13) and (19), or (27), (27) would not. This requirement is important for token preprocessing, including padding and truncation, as well as for embedding both tokens and labels.

5.2. Custom token vocabularies

This section describes how we build a token vocabulary from the given datasets (see Section 6.1.2). The primary motivation behind our process is to create various vocabularies and identify those most suitable for the KGE task. The requirements we consider for this task are applicability and KG-tailored. Based on this, we first ensure that all entities and relation types contain textual labels. We then use these labels to build a corpus. Next, we apply the byte-level BPE algorithm to construct a token vocabulary from the corpus. This alone ensures that the vocabulary satisfies the applicability requirement, as outlined in Section 5.1. As part of this process, we refined the byte-level tokenizer to better match knowledge graph data.

For the first step, we prioritize using existing labels, as is the case for the NELL dataset. While FB15k-237 and WN18RR provide textual labels for relation types, we had to replace the entity labels with previously generated ones shared by others [1]. For FB15k-237, the labels were generated by matching the Freebase IDs of entities with those in Wikidata. For WN18RR, the labels were generated by matching entities with possible textual labels and then keeping only the entities for which the known edges are valid.

In the second step, our goal is to build a corpus that best represents the KG. For this, we build the corpus from a set of edges extracted from a list of available KGs, which we refer to as the **KG pool**. At its simplest, the KG pool contains only the training triples from the dataset used. For our work, however, we use three variants of the KG pool. In *KG-custom*, we only use the training triples, and if available, the inference graph (see Section 2.1). In *KG-pretrained*, we use triples from unrelated KGs, that is, KGs that did not come from the same dataset. In *KG-finetuned*, we use all the triples of unrelated KGs, and triples from the training and inference graph. We selected these pools to create a gradient ranging from corpora that represent a specific knowledge graph to those that represent the KG space as a whole.

The set of edges extracted from the KG pool is then transformed into a list of labels that contains all possible entities and relation types. For this transformation, we use one of three variants, which we collectively refer to as the **corpus input**. Each input type emphasizes a different aspect. In $V \cup R$, we use the set of entities and relation types. The motivation behind this input type is to generate a corpus that treats all entities and relation types equally. In E, we use the set of edges, resulting in

[1] `https://github.com/villmow/datasets_knowledge_embedding`

duplicate entities and relation types. This input type shows a bias towards frequent entities and relation types. In *random walks*, we generate at least one walk of up to five entities and five relation types. One walk is generated for each entity and relation name, resulting in complete coverage. This input type shows a bias towards entities and relation types that are in a dense area of the graph.

Next, the corpus is fed into a slightly modified BPE tokenizer (see Section 2.2). These modifications consist of two changes specifically tailored to knowledge graph data. First, the corpus is represented as frequency counts for each label. With the way the corpora are created, this drastically reduces the time it takes to generate vocabularies, especially for E and *random walks*. Second, we added a **tiebreaker** for equally frequent token pairs. During the tokenization process, tokens are merged based on the frequency of pairwise occurrences. Usually, BPE behavior in such cases is not explicitly stated, as equally frequent pairs are rare in the large datasets commonly used in language tasks. However, the number of words in our corpora is comparatively small, especially $V \cup R$, which does not contain any duplicates. We therefore implemented two tiebreakers, *ascending size*, which tends to result in smaller tokens, and *descending size*, which tends to result in larger tokens. Finally, the vocabulary size is chosen between 50 000 and 100 000 tokens, depending on the number of words $|V \cup R|$. A smaller vocabulary can be extracted by truncating tokens until a desired size is reached. For this work, the final vocabulary size is governed by the hyperparameter **vocabulary size**. The total number of entities and relation types $|V \cup R|$ is multiplied by the vocabulary size hyperparameter, resulting in larger datasets using larger vocabularies.

5.3. Token length normalization

This section describes how the tokenizations of labels are processed in preparation for embedding, with a focus on preserving faithfulness. Any token length normalization beyond padding impacts the faithfulness of tokenizations, as removing tokens will inadvertently lose information. The following strategies aim to reduce the impact of token normalization.

In the original BYTE implementation, tokenizations are padded to match the length of the longest example across train, validation, and test data. As a result, no truncation is applied. We introduce five different strategies for normalizing tokenizations, collectively referred to as **truncation strategies** from this point forward. All truncation strategies include padding to match the chosen normalization length. *Last first* removes tokens from the end, following standard truncation practice. *Ascending and descending size* remove tokens based on token length, from smallest to largest and vice versa. *Ascending and descending rank* remove tokens based on their frequency rank in the vocabulary. These strategies aim to remove the least important tokens first, preserving key properties as much as possible. The normalization length for each strategy is set by the parameter **forced truncation**, which sets the normalization length to the nth percentile of the lengths of all tokenizations from elements of the training data. By default, a value of 100 corresponds to the maximum length observed in training data, while 50 corresponds to the median.

5.4. Matrix reduction

This section describes alternatives to the linear function used to combine token embeddings. Since this operation reduces a sequence of token embeddings to a single label embedding, we refer to it as **matrix reduction**. Matrix reduction impacts only the combination of token embeddings and does not influence how entity or relation type embeddings are utilized beyond the embedding layer. The choice of matrix reduction has a significant impact on the faithfulness of the label embedding. As an example, a poorly chosen method might result in two semantically distinct entities having similar embeddings. Consequently, while approaches such as one-hot vectors or token occurrence vectors (e.g., bag-of-words) are applicable, they discard too much information, which conflicts with our requirement for faithfulness.

In this work, we explored the use of Recurrent Neural Networks (RNNs) instead of a linear mapping (see Section 2.3). These networks are commonly used to process sequential input of variable length, including linguistic tasks. This aligns well with the BYTE approach, as BPE creates variable-length lists of tokens for each input. As a result, they allow padding and truncation to be skipped entirely, preserving information and differences more effectively. We tested three common types of RNNs: Simple RNNs, Gated Recurrent Units (GRUs), and Long Short-Term Memory (LSTM) networks. Each RNN receives an unpadded sequence of token embeddings as input. The final hidden state is used as the embedding representing the entity or relation type label.

5.5. Multiple tokenizations

This section describes a method for leveraging smaller unmerged tokens that would otherwise remain unobserved during training. This method generates multiple copies of each edge, each utilizing a different tokenization. It has a significant impact on the faithfulness of token embeddings, as embeddings of unexplored tokens do not preserve key properties or differences.

The core aspect of this modification is to increase the amount of training data by constructing up to n different variants of each edge, governed by the parameter **multiple bpe encodings**, referred to as **multitoken** from this point on. These variants are chosen based on the maximum allowable token length. By default, this parameter is set to $n = 1$, representing the standard behavior in which each edge is tokenized a single time using the full vocabulary. For values $n > 1$, each edge is tokenized n times, with the allowed token length increasing each time. Duplicate tokenizations are discarded.

In addition to the prior modification, we also implemented an additional loss term for regularization, used explicitly for a multitoken value greater than one. The underlying parameter for this loss term is **multiple bpe loss**, referred to as **multiloss** from this point on. When multiloss is enabled, training penalizes discrepancies between embeddings of the same entity or relation type. This is achieved by averaging embeddings from all tokenizations of the same label and minimizing the mean squared error between each embedding and this average. The total loss after each epoch is given by the original loss $\ell(\hat{y}, y)$ plus the regularization term:

$$\mathcal{L} = \ell(\hat{y}, y) + \sum_{l \in V \cup R} \sum_{i=1}^{n} \left\| \mathbf{e}_l^{(i)} - \bar{\mathbf{e}}_l \right\|^2 \quad \text{where} \quad \bar{\mathbf{e}}_l = \frac{1}{n} \sum_{i=1}^{n} \mathbf{e}_l^{(i)}$$

Here, the set $V \cup R$ contains all entity and relation labels. The parameter n is the multitoken value, indicating the number of tokenizations per label. $\mathbf{e}_l^{(i)}$ is the embedding of the i-th tokenization of label l. $\bar{\mathbf{e}}_l$ is the average embedding over all n tokenizations of label l. $\| \cdot \|^2$ denotes the squared Euclidean norm, measuring the squared distance between vectors. This regularization encourages embeddings of different tokenizations of the same label to remain close, improving faithfulness without altering the model.

6. Evaluation

This evaluation focuses on addressing research questions **RQ1** and **RQ3**. To evaluate baseline performance and assess the impact of our proposed approach and its variants, we used a diverse set of datasets. By systematically exploring multiple hyperparameter configurations, we were able to analyze the contribution of each component in isolation.

This section is structured as follows. In Section 6.1, we present the experimental setup. In Section 6.2, we present and discuss the results of our benchmarks.

6.1. Experimental setup

This section describes the experimental setup used to evaluate BYTE and our proposed approach, covering hardware and software specifications, datasets, model configurations, and the hyperparameter tuning procedure.

In Section 6.1.2, we present the different datasets employed and explain our selection criteria. In Section 6.1.3, we describe the model configurations and training procedures. In Section 6.1.4, we detail our approach to hyperparameter search.

6.1.1. Hardware and software

Our work builds on the DICE Embeddings framework (dicee)[2], which we modified according to our approach described in Section 5. These modifications are available

[2] `https://github.com/dice-group/dice-embeddings`

in our GitHub repository[3], along with a custom implementation of a BPE-based tokenizer and standalone evaluation scripts. All experiments were conducted using Python 3.11.

The core Python packages used in both the DICE framework and our independent components are:

- **PyTorch**[4] 2.6.0,
- **Pandas**[5] 2.2.3, and
- **tiktoken**[6] 0.9.0.

PyTorch serves as the backbone for model training, supporting execution on both multiple CPU cores and GPUs. Pandas is extensively used for preprocessing graph data and aggregating benchmark results. Finally, tiktoken enables efficient segmentation of textual labels into subword units.

Computational resources from the high-performance computing cluster HoreKa were used for the evaluation. The hardware specifications of the compute nodes are as follows:

- **CPU:** 2× Intel Xeon Platinum 8368
- **GPU:** 4× NVIDIA A100-40
- **GPU memory:** 40 GB per GPU
- **Main memory:** 512 GB

The first two evaluation settings, along with the exploratory experiments, used one GPU, 38 CPU cores, and 124 375 MB of main memory. For the remainder of the evaluation, the number of CPU cores was reduced to one.

6.1.2. Datasets

All evaluations took place on commonly used datasets [3, 30]. Specifically, we use (fully-)inductive splits based on NELL-995 [31], FB15k-237 [32], and WN18RR [24]. The inductive splits of NELL-995, FB15k-237, and WN18RR were introduced in [3], and the fully-inductive split of NELL-995 was introduced in [33]. In addition to those datasets, we also included partially inductive splits of NELL-995 used in BYTE and introduced in [34].

[3] https://github.com/Fattehh/BPE-for-Knowledge-Graph-Embeddings
[4] https://pytorch.org/
[5] https://pandas.pydata.org/
[6] https://github.com/openai/tiktoken

Table 2: **Basic dataset statistics.** The table presents the distribution of entities V, relation types R, and triples E.

Dataset	$\lvert V_{\text{train}}\rvert$	$\lvert R_{\text{train}}\rvert$	$\lvert E_{\text{train}}\rvert$	$\lvert V_{\text{test}}\rvert$	$\lvert R_{\text{test}}\rvert$	$\lvert E_{\text{test}}\rvert$
FB V3	3648	215	17986	980	128	865
NELL V1	3103	14	4687	84	7	100
NELL V4	2092	76	7546	630	47	731
NELL-h25	64634	172	122618	10245	172	9187
NELL-h75	26219	57	59135	4600	57	4389
NL-25	4396	106	17578	598	75	744
WN V2	6988	10	15262	710	9	441
WN V4	3885	9	7940	2273	9	1429

We selected a total of seven splits and performed a few additional experiments on an eighth. All of these splits include triples that contain at least one unseen element. The selected splits consist of the inductive datasets FB V3, NELL V1, NELL V4, WN V2, and WN V4; the fully-inductive split NL-25; and the mostly transductive splits NELL-h75 and NELL-h25, with the latter used only in a few experiments. The mostly transductive splits contain between 20 and 30% edges with unseen entities. Furthermore, all inductive datasets contain inference graphs.

For an overview of basic dataset statistics, such as the number of entities, see Table 2. Table 3 presents statistics related to the inductive nature of the datasets, where $V_{\text{unseen}} \subseteq V_{\text{test}}$, $R_{\text{unseen}} \subseteq R_{\text{test}}$, and $E_{\text{unseen}} \subseteq E_{\text{test}}$. This table also includes the mean minimum Levenshtein distance, a metric designed to measure the degree of difference between unseen elements in the test graph and elements in the training graph. It is computed by averaging the minimum Levenshtein distance between each test graph element and the most similar element in the training graph. Additionally, an overview of all considered splits is provided in Table 11.

Our primary selection criterion was variety. Accordingly, we selected at least one split from each type of dataset we identified. First, **FB15k-237**, a subset of the manually curated Freebase knowledge graph. Second, **NELL-995**, a subset of the automatically constructed NELL knowledge graph. Finally, **WN18RR**, derived from the English lexical database WordNet. For the specific splits, we selected those that exhibited notable differences in the distribution or similarity of elements between the training and test data. Among the **FB15k-237** splits, we found only statistical differences that were roughly proportional to the size differences between them. We therefore selected FB V3 to enhance the overall variety of our split selection. It offers both a high number of relation types and the lowest dissimilarity among the FB splits, while still exhibiting the second-highest dissimilarity among all chosen splits. Among the **NELL-995** splits, we observed greater variety, which is why most of our selected splits come from this dataset. First, NELL V1 and V4 show a significant difference in dissimilarity. Second, NL-25 is the fully-inductive split with

Table 3: **Dataset statistics concerning inductivity.** The table presents counts of unseen entities V_{unseen}, relation types R_{unseen}, inductive triples E_{unseen}, and the mean minimum Levenshtein distance, which measures the differences between seen and unseen elements.

Dataset	$\lvert V_{\text{unseen}} \rvert$	$\lvert R_{\text{unseen}} \rvert$	$\lvert E_{\text{unseen}} \rvert$	Mean min. Levenshtein
FB V3	970	0	865	5.396
NELL V1	84	0	100	1.659
NELL V4	629	0	731	7.388
NELL-h25	2831	0	2459	1.435
NELL-h75	978	0	950	1.051
NL-25	598	35	744	4.851
WN V2	710	0	441	2.650
WN V4	2273	0	1429	2.907

the least dissimilarity and the third highest dissimilarity among the selected splits. Third, the mostly transductive splits NELL-h25 and NELL-h75 show very low dissimilarity. Finally, among the **WN18RR** splits, we chose WN V2 and WN V4 for their significant difference in the ratio between training and test entities.

6.1.3. Model configuration and training

All experiments in this work utilize a modified version of the training setup described in [10], which includes the cross-entropy loss function, KvsAll scoring technique, and Adam optimizer. No additional regularization techniques were applied, except for the multiloss regularization described in Section 5.5.

On this basis, we conducted two types of experiments. The first type consists of short experiments aimed at identifying the optimal hyperparameter configuration for the second type of experiments. Each of these experiments is performed once, without early stopping, typically for 100 epochs. For experiments using RNNs for matrix reduction, the number of epochs is doubled to 200. For a breakdown on all short experiments, see Section 6.1.4.

The long experiments correspond to full training and evaluation runs performed on established benchmark datasets. These benchmarks are used to evaluate the performance of BYTE and our approach, including its variants. For these experiments, we begin by fixing one or more hyperparameters. We then select the short experiment with the best validation MRR among those using the fixed hyperparameter values. This experiment is then repeated five times, using different seeds each time. Each experiment runs for a minimum of 500 to a maximum of 1000 (2000 in the case of RNNs) epochs. Early stopping is performed by calculating the MRR on the validation data every ten epochs. Training is terminated if the validation MRR does not improve for ten consecutive evaluations.

Table 4: **Evaluated variations of BYTE** with associated hyperparameters and their tested values.

Variation	Hyperparameter	Values
Vocabulary	KG pool	KG-custom, KG-pretrained, KG-finetuned, GPT-2
Matrix reduction	Matrix reduction	Linear, RNN, GRU, LSTM
Truncation	Truncation strategy	last_first, asc_size, desc_size, asc_rank, desc_rank
	Forced truncation	90, 95
Multitoken	Multitoken	2, 3, 4
	Multiloss	True, False

In total, we performed one set of long experiments for each variation of BYTE (see Table 4). First, we fixed the KG pool parameter to evaluate the baseline performance of BYTE and our proposed approach using different token vocabularies. Second, we fixed the matrix reduction hyperparameter, evaluating the performance of different matrix reduction methods. Third, we fixed the truncation strategy and the forced truncation parameter separately, allowing us to evaluate the impact of the different truncation strategies and the amount of truncation. For our final set, we fixed the multitoken hyperparameter and conducted one set of runs with multiloss and one without. This enables us to assess the benefits of exploring previously unseen tokens and the effects of encouraging embeddings of the same label to be more similar. Whenever possible, experiments were reused across benchmarks if the corresponding hyperparameter configuration matched.

6.1.4. Hyperparameter search

Our hyperparameter search was conducted through multiple smaller grid searches. While a full grid search would have enabled evaluation of all possible hyperparameter combinations, its computational cost grows exponentially with the number of hyperparameters. In our case, we use four hyperparameters to determine the chosen vocabulary, three to determine the model behavior, and five for the modifications suggested in this work—for a total of 12. Therefore, a full grid search for three values per parameter would require $3^{12} = 531\,441$ experiments for each dataset. Instead, we performed our search in three steps.

Table 5: **Parameter search for vocabularies evaluated for integration into ByTE.** Each row represents a grid search across the dimensions represented by the columns.

Vocabulary parameters				Model parameters		
KG pool	Corpus input	Tie breaker	Voc. size	Model	Dims	L. rate
KG-custom	VandR E randomWalks	asc_size desc_size	0.5 1.0 2.0	Keci ComplEx	16 32 64	0.1 0.01
KG-pretrained KG-finetuned	VandR	asc_size desc_size	0.5 1.0 2.0	Keci ComplEx	16 32 64	0.1 0.01
GPT-2				Keci ComplEx	16 32 64	0.1 0.01

Table 6: **Parameter search for the different variants of ByTE.** Each row represents a grid search across the dimensions represented by the columns. Best values are taken from previous searches.

	Modification parameters			Model parameters		
Vocabulary	Matrix reduc.			Model	Dims	L. rate
Best vocab	RNN GRU LSTM			Keci ComplEx	16 32 64	0.1 0.01

Vocabulary	Matrix reduc.	Trunc. strat	Forced	Model	Dims	L.rate
Best vocab	Best reduc.	last_first asc_size desc_size asc_rank desc_rank	90 95	Keci ComplEx	16 32 64	0.1 0.01

Vocabulary	Matrix reduc.	Multitoken	Multiloss	Model	Dims	L.rate
Best vocab	Best reduc.	2 3 4	False	Keci ComplEx	16 32 64	0.1 0.01

In the first step, we searched for the best hyperparameter configurations that did not involve modifications of the BYTE approach. This includes all vocabulary hyperparameters and the hyperparameters used for the underlying BYTE model. The Table 5 shows the hyperparameter values we searched for. Each row represents a full grid search on all values of each column, while each column represents the dimensions of the grid search. For example, the first row performs $1 \cdot 3 \cdot 2 \cdot 3 \cdot 2 \cdot 3 \cdot 2 = 216$ experiments per dataset. This amounts to a total of 372 experiments per dataset, or 2 604 experiments for the seven datasets. Each of those experiments was performed with 100 epochs and without early stopping. This search enabled us to select the best vocabulary for each dataset. These hyperparameters are used to perform our first benchmark for the unmodified BYTE approach. Additionally, the vocabulary hyperparameters were reused in the second partial grid search.

In the second step (As shown in the first row of Table 6), we utilized the best vocabulary hyperparameters to search for the best RNN for the matrix reduction, conducting a total of 252 experiments. Each of those experiments was conducted with 200 epochs and without early stopping. In the third step (represented by the second and third rows of Table 6), we utilized the best vocabulary and matrix reduction to search for the best truncation and multitoken parameters, conducting a total of 1 092 experiments. Each of those experiments was conducted with 100 or 200 epochs and without early stopping.

6.2. Results

All experiments in this work are evaluated on the train, validation, and test data. All hyperparameter configurations and results are available on GitHub[7].

This subsection is structured as follows. In Section 6.2.1, we discuss the aggregated results of all our benchmarks. In Section 6.2.2, we present the results of our benchmark on different vocabularies.

6.2.1. Aggregate result analysis

For each dataset, between 18 and 22 hyperparameter configurations were tested, each comprising five long experiments. Two exceptions apply: two multitoken configurations on the NELL-h75 benchmark were stopped early after only three experiments each, and no more than seven hyperparameter configurations were evaluated using the NELL-h25 dataset, all of which were stopped at one or two experiments.

Fig. 2 shows the aggregated results of all configurations. Benchmarks are sorted based on the similarity between training and test graph labels, measured by the mean minimum Levenshtein distance (see Table 3). The MRR for each configuration is calculated as the mean of the MRRs from its associated experiments. State-of-the-art (SOTA) results for the predominantly transductive datasets, NELL-h25 and NELL-h75, were taken from [34], while results for the other datasets were sourced from [4]. Results for BYTE [10] were only available for NELL-h25 and NELL-h75.

[7] https://github.com/Fattehh/BPE-for-Knowledge-Graph-Embeddings

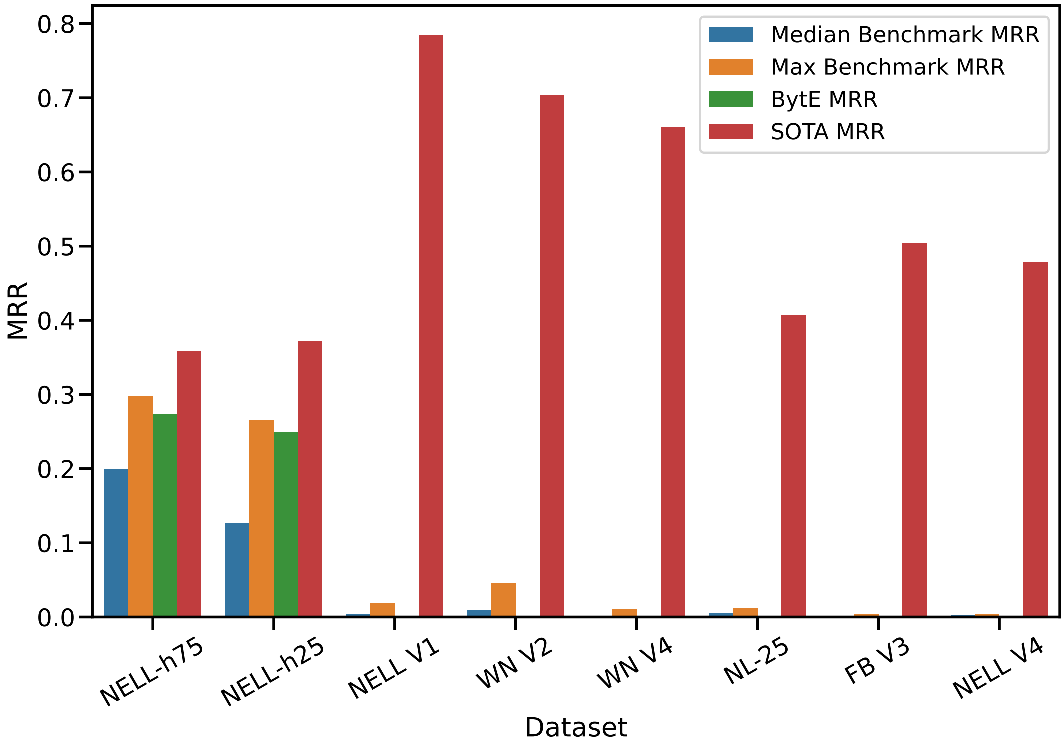


Figure 2: **The best hyperparameter configuration from our experiments outperforms the original BYTE, showing strong results on mostly transductive benchmarks, but falling significantly behind the state-of-the-art (SOTA) on inductive benchmarks.** This figure shows mean reciprocal rank (MRR) performance across our selection of benchmarks. The bars represent the median and maximum MRRs across all experiment configurations of this work, alongside results from BYTE and the SOTA.

Overall, it is clear that BYTE performs poorly across all inductive datasets considered. This is especially evident in datasets where test labels lack similar counterparts in the training graph. Mean MRRs of configurations on these datasets exhibit similarly low performance, with no single configuration clearly outperforming the others. This indicates that BYTE is incapable of drawing any generalizable information from the training data for these datasets. One possible explanation is that the training and test labels are entirely unrelated, making label-based embeddings insufficient for inductive link prediction on these datasets. Alternatively, the poor performance may be caused by limited token overlap: dissimilar labels likely result in divergent tokenizations, leading to a lack of shared token representations. Finally, BYTE does not leverage any information from the inference graph, which may further limit its ability to generalize in this setting. Notably, recent research suggests that the inference graphs used in current benchmark datasets are poorly constructed, allowing non-learnable heuristics that ignore relational information to achieve MRR scores exceeding 40% on datasets such as WN V2, WN V4, and FB V3 [35]. This highlights that methods that do not utilize the inference graph operate at a significant disadvantage.

In contrast, BYTE achieves substantially higher performance on the predominantly transductive datasets NELL-h25 and NELL-h75 than on the inductive ones. While these datasets naturally exhibit lower mean minimum Levenshtein distances between their training and test labels, the magnitude of the performance improvement relative to the next most similar inductive dataset is disproportionate. However, two points are worth noting. First, the SOTA results are achieved using the transductive KGE method MuRP [34], which can be used in the BYTE approach. Since applying BYTE before transductive KGE methods has led to improved performance over using the KGE methods alone on both NELL-h75 and NELL-h100, it is plausible that using BYTE in conjunction with MuRP would yield similarly improved results. Second, the experiments on NELL-h25 were performed using models that achieved extremely low results (mean MRR of 0.003) in the evaluation by [10]. All of our configurations achieved higher mean MRRs, with the best reaching a value over 80 times greater and even outperforming their strongest results obtained using more suitable models. As such, our proposed modifications resulted in lower instability of the results, thereby preventing extreme underperformance.

6.2.2. BYTE with different vocabularies

For each dataset, we evaluated four hyperparameter configurations, each of which was tested across five long experiments. Each configuration is the best combination of parameters found for the four vocabulary types: KG-custom, KG-finetuned, KG-pretrained, and GPT-2. All hyperparameter configurations used are listed in Table 12, and their results are shown in Table 13. Fig. 3 presents the results across the selected benchmarks, sorted by mean minimum Levenshtein distance (see Table 3). The MRR score for each configuration is the mean MRR across the five experiments.

From the results, no clear advantage is evident in using custom vocabularies. This is particularly notable in the case of NELL-995-h75. During initial experimentation with BYTE, we observed improved performance on purely transductive benchmarks, particularly when KG-custom was in use (see Fig. 7 in Appendix B). The reduced performance on NELL-995-h75 implies either an incompatibility with the dataset or the setting. Adding to this, the results on WN V2 show the opposite effect, where KG-pretrained achieved better results on an inductive dataset. This implies that there is a tradeoff between how well a vocabulary is suited for transductive and inductive link prediction. The more general or unrelated the vocabulary, the worse it performs on a transductive split, and similarly, the better it performs on inductive splits. However, this hypothesis is not fully supported by the data, as the results on NELL V4 do not support this.

One explanation for this effect is coincidence. While the experiments on KG-

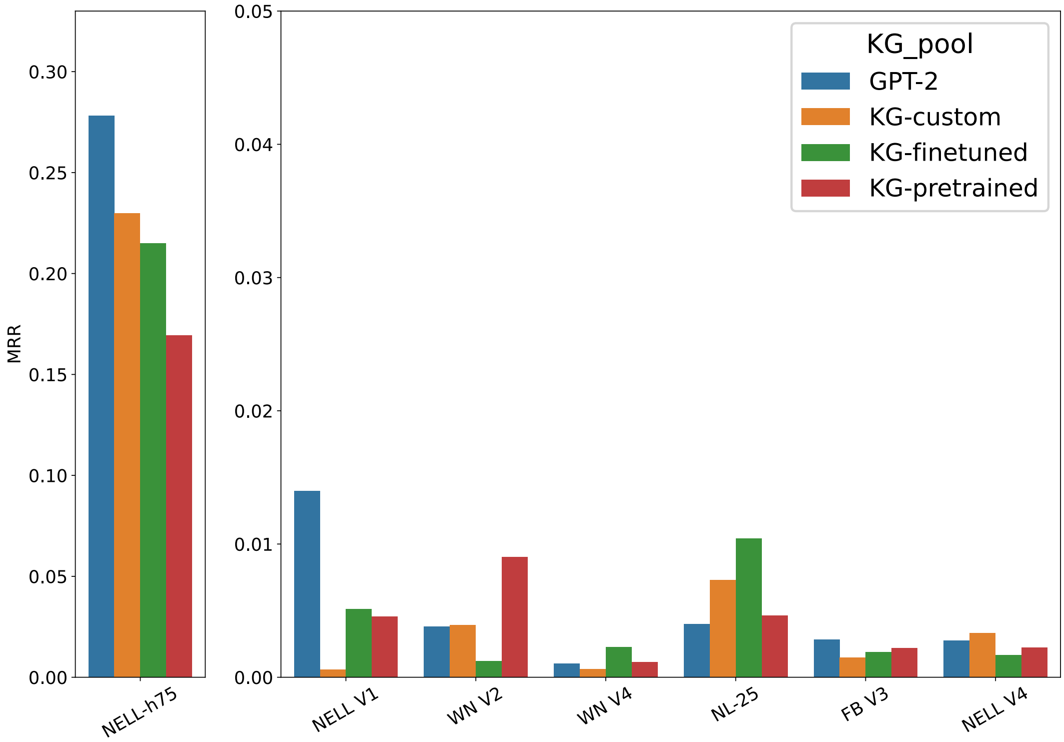


Figure 3: **Vocabulary choice strictly depends on the chosen dataset. Some benefit from well-fitting vocabularies while others benefit from more general ones like GPT-2.** This figure shows mean reciprocal rank (MRR) performance across our selection of benchmarks. The bars represent the median and maximum MRRs across the four different vocabularies used in this work.

pretrained yield the best mean results, they are also among the few sets of experiments where the standard deviation exceeds the mean MRR. At the same time, the standard deviation of the GPT-2 experiments on NELL-h75 is significantly lower, despite yielding the best mean value. Another explanation, especially when considered alongside the results discussed in Section 6.2.5, is a lack of token exploration. In particular, the NELL-h75 results can be attributed to it being the largest dataset and, consequently, having the largest vocabulary. This is a direct result of our approach, which employs relative vocabulary sizes. Large vocabularies tend to use fewer tokens per entity or relation, which limits the exploration of smaller subword tokens. To give an extreme example, if the vocabulary is sufficiently large, all entity and relation type labels from the training data may be stored as individual tokens, meaning only those tokens are ever being observed. The tokenization of any unseen label that is not a combination of two or more seen labels will therefore contain unseen tokens, which negatively affects performance. This effect closely resembles

Table 7: **Custom vocabularies led to reductions in training time by a factor of up to 2.4.** The table reports training speed relative to using the GPT-2 vocabulary, which serves as the baseline (100%).

Dataset	KG-custom	KG-finetuned	KG-pretrained
FB V3	177.0%	153.3%	137.3%
NELL-h75	102.0%	98.2%	95.3%
NL-25	115.9%	94.6%	138.8%
NELL V1	242.3%	107.2%	175.3%
NELL V4	117.7%	111.2%	171.1%
WN V2	144.7%	116.0%	116.3%
WN V4	156.7%	54.3%	72.0%

overfitting, suggesting that in inductive settings, the standard BYTE approach benefits most from vocabularies that do not result in tokenizations with unique tokens. In the context of the BPE tokenizer, this means that merging any token pair might result in a performance loss.

However, as shown in Table 7, except for NELL-h75, custom token vocabularies generally reduced computational costs. For this reason alone, their use is justified in transductive settings, where handling unseen entities is not a primary concern. Additionally, the vocabulary size was chosen relative to the number of entities and relation types. Selecting an absolute size instead could enable more targeted reductions in computational cost.

6.2.3. BYTE with matrix reduction

For each dataset, we evaluated at least three hyperparameter configurations, each of which was tested across five long experiments. Each configuration is the best combination of parameters found for the three RNN types used for matrix reduction: GRU, LSTM, and RNN. Additionally, we show the best result from Section 6.2.2. For the datasets for which more than three configurations were evaluated, we displayed the best result for each RNN type, while also indicating the range of results with a gray bar. All hyperparameter configurations used are listed in Table 14, and their results are shown in Table 15. Fig. 4 presents the results across the selected benchmarks, sorted by mean minimum Levenshtein distance (see Table 3). The MRR score for each configuration is the mean MRR across the five experiments.

While the results do not show improvements across all benchmarks, the inclusion of RNNs has merit. In particular, it seems to smooth the transition from token embeddings to full representations, yielding modest gains on NELL-h75 and WN V2. However, these gains are relatively minor, and other datasets do not show such improvement. Notably, the standard deviation consistently decreases, indicating improved stability. Overall, the use of RNNs contributes to more stable performance,

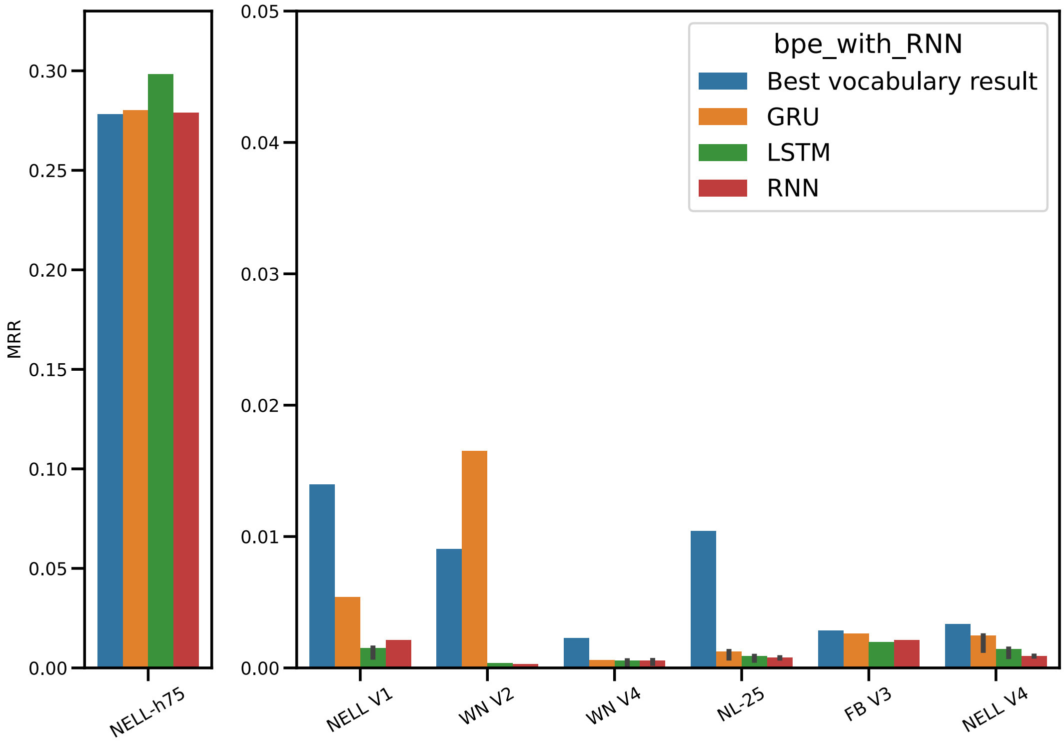


Figure 4: **While dataset-dependent, RNNs can lead to clear improvements in MRR scores.** This figure shows mean reciprocal rank (MRR) performance across our selection of benchmarks. The bars represent the median and maximum MRRs across the three different types of matrix reduction used in this work. Additionally, results from Fig. 3 are used as a baseline.

with the potential for minor accuracy improvements in certain settings.

One explanation for this effect is the lack of padding. Depending on how the embedding for padding is handled, it may cause fluctuations in the prediction process. This is particularly relevant in cases of extremely low performance (e.g., below 1% MRR), such as those observed in inductive settings, where even minor inconsistencies may lead to noticeable variance. Another explanation is that the RNNs are implicitly trained to complement the architecture they are integrated with. That is, rather than functioning purely as independent sequence encoders, they may adapt during training to produce representations that are better suited to the downstream model, whether by aligning token sequences more closely with expected input distributions or by smoothing inconsistencies in token-level information.

While these effects are generally positive, they do have a heavy impact on performance, as shown in Table 15. In the most extreme case, we observed a slowdown by a factor of 18 when using GRU on WN V2. Although this particular instance may be attributable to hardware issues, other datasets also exhibited substantial reductions

Table 8: **RNNs can both reduce and increase the training time of BYTE.** The table reports training speed relative to using a linear function, which serves as the baseline (100%).

Dataset	GRU	LSTM	RNN
FB V3	26.1%	26.3%	26.1%
NELL-h75	48.5%	34.2%	50.8%
NL-25	77.8%	146.3%	145.1%
NELL V1	84.0%	115.8%	85.3%
NELL V4	84.6%	106.0%	117.6%
WN V2	5.5%	32.8%	134.0%
WN V4	120.5%	124.7%	119.1%

in speed. However, the increased training stability often led to earlier convergence and increased the likelihood of early stopping. As a result, computational costs are highly variable, with noticeable speedups when models converge early and pronounced slowdowns when they do not.

6.2.4. BYTE with truncation

For each dataset, we evaluated between five and seven hyperparameter configurations, each of which was tested across five long experiments. Each configuration is the best combination of parameters found for the seven parameter values forced@90, forced@95, asc_rank, asc_size, desc_rank, desc_size, and last_first. Each hyperparameter configuration was only used five times, without further repetition. We additionally show the best result from Section 6.2.2. For any hyperparameter value with multiple associated configurations, we present the best result, while also indicating the range of results with a gray bar. All hyperparameter configurations used are listed in Table 16, and their results are shown in Table 17. Fig. 5 presents the results across the selected benchmarks, sorted by mean minimum Levenshtein distance (see Table 3). The MRR score for each configuration is the mean MRR across the five experiments.

Unlike previous variants, this variant achieved only one significant performance increase, in the case of WN V4. However, both it and any other apparent performance increases are overshadowed by a significant standard deviation, reflecting high inconsistency in performance. These increases have two possible explanations. First, coincidence, as both results are well within the range of previous results, especially the WN V2 results, with a deviation larger than the mean. This effect would be strengthened if the seen tokens are removed, as the model would be more random in that case. Second, the chosen strategy might have inadvertently removed unseen tokens, resulting in an actual, albeit minor, improvement. However, neither of those results is major or reliable enough.

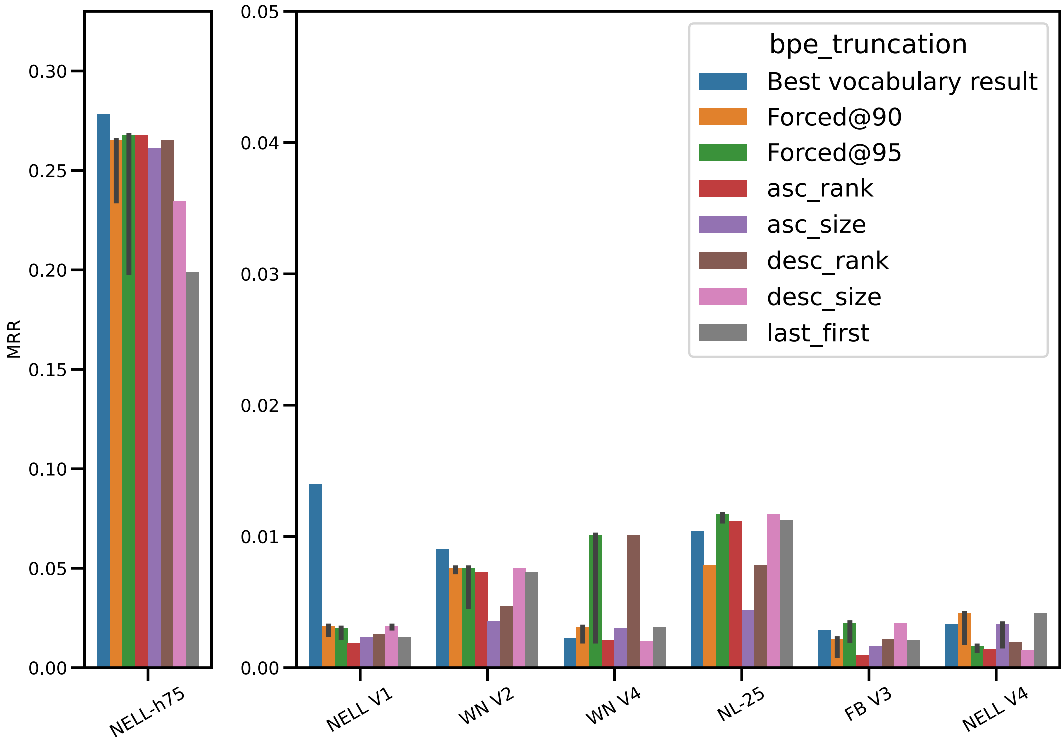


Figure 5: **Truncation shows mixed results, with no strategy showing consistent results.** This figure shows mean reciprocal rank (MRR) performance across our selection of benchmarks. The bars represent the median and maximum MRRs across the five different truncation strategies and the two different normalization lengths used in this work. Additionally, results from Fig. 3 are used as a baseline.

Nevertheless, the other results offer valuable additional insights. In particular, last_first performed noticeably worse in the case of NELL-h75, while performing similarly in the other NELL splits. This is most likely caused by the predominantly transductive nature of NELL-h75 and its hierarchical label structure. Since the final segment of each label tends to be the most distinctive, removing these last tokens leads to more similar embeddings, making it more difficult for the model to distinguish between labels. In a transductive setting, this is harmful, as it treats similar entities and relation types as identical. In an inductive setting, however, this is a benefit.

Another anticipated benefit of label shortening was a reduction in computational cost. Shorter labels were expected to reduce sequence lengths, thereby simplifying model processing and leading to faster training times. However, the data shown in Table 9 does not support this expectation. We suspect this discrepancy is due to an issue in the experimental setup. After the unmodified and linear benchmarks, we observed that excessive CPU usage yielded minimal benefit, mainly limited to faster data loading, which should not have significantly affected the training time. However, the validation data used for early stopping was read from disk rather than being preloaded into memory. When accounting for the time spent loading validation data, the actual training times were at least comparable to, if not shorter than, the original setup. The factor of speedup for NELL-h75 supports this explanation, as its training time dominates over data loading time.

Table 9: **Truncation led to a general increase in training time of BYTE.** The table reports training speed relative to experiments without normalization, which serve as the baseline (100%).

Dataset	90	95	asc_rank	asc_size	desc_rank	desc_size	last_first
FB V3	75.45%	102.32%	45.97%	111.73%	104.93%	111.28%	102.32%
NELL-h75	100.89%	100.68%	101.26%	100.88%	100.10%	100.48%	101.09%
NL-25	51.08%	52.66%	50.78%	55.86%	51.08%	48.63%	56.16%
NELL V1	62.43%	72.66%	75.23%	76.30%	62.32%	63.18%	77.03%
NELL V4	53.01%	65.74%	59.90%	59.70%	50.79%	69.44%	60.77%
WN V2	51.37%	53.40%	49.11%	56.05%	55.97%	53.77%	48.61%
WN V4	65.98%	63.07%	78.97%	48.74%	78.05%	81.59%	49.14%

6.2.5. BYTE with multitoken

For each dataset, we evaluated six hyperparameter configurations, each of which was tested across five long experiments. The only exception was NELL-h75, where the results for quadruple tokenizations were conclusive after just three runs. Each configuration corresponds to the best combination of parameters selected from the six possible pairs formed by the values 2, 3, 4 and False, True. We additionally show the best result from Section 6.2.2. All hyperparameter configurations used are listed in Table 18, and their results are shown in Table 19. Fig. 6 presents the results across the selected benchmarks, sorted by mean minimum Levenshtein distance (see Table 3). The MRR score for each configuration is the mean MRR across the five experiments.

The results give valuable information on the characteristics of BYTE. First, multiple tokenizations are most useful for inductive datasets with a low mean minimum Levenshtein distance of the test labels. That is, if there is a seen label that is similar to an unseen one, BYTE performs better if it sees multiple tokenizations of both. This strongly implies that BYTE does not explore enough tokens. Second, the multiloss regularization we utilized was too aggressive, drastically reducing performance in the mostly transductive benchmark NELL-h75. However, results on benchmarks other than NELL-h75 and WN V2 give some incentive to embed different tokenizations of a label into similar embeddings. Third, since the baseline outperforms all multiple-tokenization variants on the NELL-h75 benchmark, it is evident that BYTE

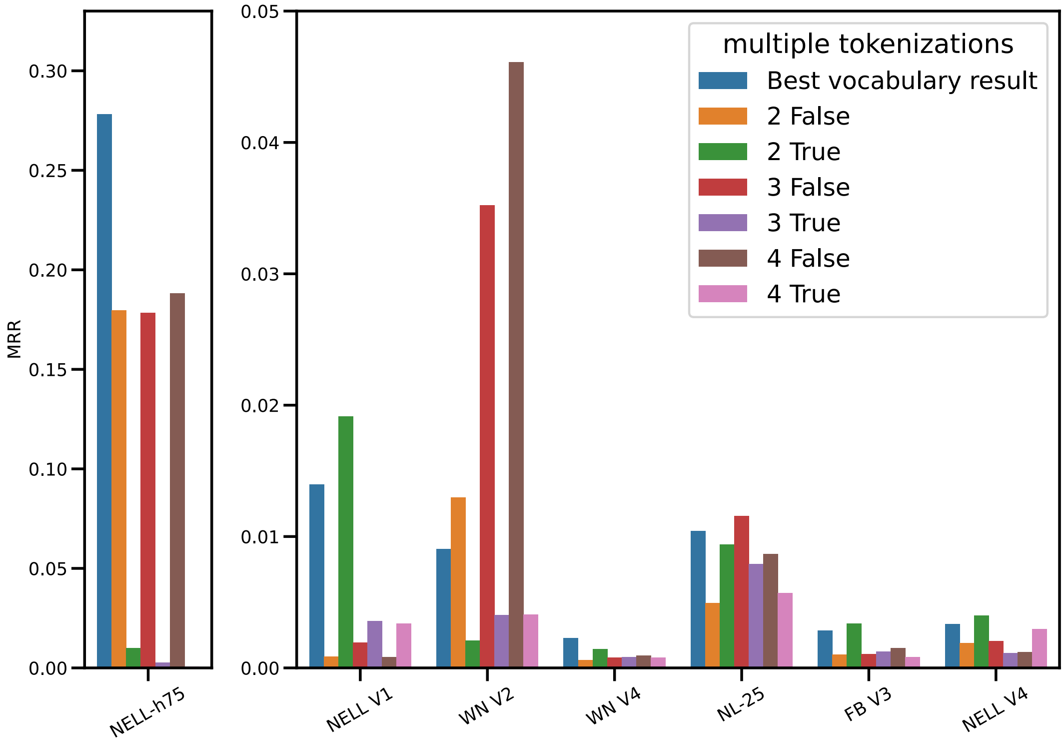


Figure 6: **With the exception of WN V4, multitoken showed an improvement in all inductive benchmarks.** This figure shows mean reciprocal rank (MRR) performance across our selection of benchmarks. The bars represent the median and maximum MRRs across the three multitoken values, with 'True' indicating enabled multiloss. Additionally, results from Fig. 3 are used as a baseline.

still favors single-token tokenizations in transductive settings.

From the above-mentioned results and those shown in both Section 6.2.2 and Fig. 5, it is clear that BYTE has difficulty correctly interpreting an arbitrary set of tokens. It works best when all tokens are previously seen.

Notably, the efficiency of BYTE is directly bound to this behavior. As shown in Table 10, exploring additional tokens has a heavy impact on performance. Even a multitoken value of two already more than doubles the time it takes to train a model.

Table 10: **Using multiple tokenizations led to a significant increase in training time across all benchmarks.** The table reports training speed relative to experiments using single tokenization per edge, which serve as the baseline (100%).

Dataset	multiloss = False			multiloss = True		
	2	3	4	2	3	4
FB V3	0.247	0.256	0.109	0.270	0.253	0.111
NELL-h75	0.420	0.224	0.139	0.408	0.224	0.148
NL-25	0.299	0.234	0.160	0.324	0.232	0.157
NELL V1	0.438	0.412	0.292	0.441	0.397	0.280
NELL V4	0.430	0.230	0.137	0.450	0.214	0.138
WN V2	0.202	0.151	0.126	0.231	0.149	0.138
WN V4	0.239	0.176	0.108	0.240	0.172	0.102

7. Conclusion

This work investigated the applicability and limitations of BYTE embeddings for inductive link prediction on knowledge graphs, focusing on systematic benchmarking of both the original approach and our proposed variants. Notably, BYTE does not assume the existence of an inference graph. Since relying on such a graph is a strong assumption, the setting without an inference graph is both more challenging and more practical. However, our results demonstrate that BYTE is unable to achieve meaningful performance in this more challenging setting.

With regard to **RQ1**, as seen in Section 6.2.1, the performance of BYTE in purely inductive settings lags significantly behind state-of-the-art methods. Although it can be argued that BYTE operates under a stricter condition—that is, inductive link prediction without access to an inference graph—no configuration of BYTE achieved a mean MRR exceeding 0.1. Our answer to RQ1 is therefore that BYTE is severely lacking in standard (fully-)inductive settings.

Regarding **RQ2** and **RQ3**, we explored several approaches to creating custom BPE vocabularies tailored for knowledge graph embeddings, as presented in Section 5.2. Despite these efforts, none of the proposed methods yielded consistent improvements across the benchmark datasets (see Table 5). A key observation is that BYTE does not train embeddings for tokens that were always merged during training. Consequently, generating a token vocabulary specific to the training set reduced the number of observed tokens during inference, leading to a decrease in performance in inductive scenarios. Our findings indicate that a custom BPE vocabulary for BYTE must strike a careful balance: it should be general enough to cover tokens likely to appear during testing, but not so general that token expressiveness is lost. A well-chosen vocabulary can improve performance; however, selecting such a vocabulary without prior knowledge of the test data is exceedingly challenging.

Regarding **RQ4**, we explored and evaluated multiple modifications at different stages of the embedding process. Changing the process of transforming token embeddings into label embeddings by incorporating RNNs has led to notable improvements in stability and moderate enhancements in a predominantly transductive setting. Adding a truncation step before embedding has shown promise in increasing generalizability, although the results are minor. Introducing token exploration by training on multiple tokenizations showed promising results; however, this modification did not work universally, and the results remain far worse than those of SOTA techniques. The best result from any of our configurations did not reach 10% of the SOTA performance on inductive benchmarks.

To summarize our findings, using embeddings of textual labels to bridge the gap between transductive and inductive link prediction is at best highly dataset-dependent. At worst, it proves entirely ineffective, offering no meaningful predictive capability. However, it remains useful for transductive settings, with a well-suited vocabulary resulting in both improved performance and reduced computational requirements. As a generalizable embedding method, it may serve as a beneficial preliminary step before applying dedicated inductive link prediction techniques.

8. Future Work

The results of our work have shown that, regardless of our modifications, BYTE does not perform well in inductive settings. While we do not consider textual labels sufficient for inductive link prediction, we identify three key areas for potential improvement:

First, BYTE does not make use of the inference graph. Although we included the inference graph in the pool used to construct the token vocabulary, relying on it solely for tokenization is likely insufficient. Instead, either the (token) embeddings or models need to be influenced by the inference data. This can be achieved by introducing additional weights, biases, or neural network layers that adapt the embeddings based on both the inference data and the current state of the embeddings or the model. Such adaptation can be implemented through various techniques, including meta-learning or neighborhood aggregation methods, as exemplified by the approaches discussed in Section 3.2.

Second, BYTE does not train on tokens other than those needed to tokenize the training data. To address this, three approaches are available. First, *improving the tokenization approach*. As shown in our work, pure BPE produces vocabularies that are overly dense, containing many tokens that are never used during the tokenization of the KG labels. One option is to change the tokenization strategy, for example, by performing BPE on triples instead of pairs of tokens. This would skip all even-length tokens, resulting in a vocabulary with roughly half as many elements. Alternatively, the vocabulary could be trimmed instead, removing tokens based on a chosen metric. Both approaches significantly reduce the number of tokens, thereby lowering the number of unexplored tokens. This may enhance the accuracy of link prediction while also accelerating the training process. Furthermore, both approaches can utilize the inference graph. For example, a specially designed tokenizer might be able to adjust a vocabulary based on the inference graph. Similarly, trimming might be done based on the inference graph. Second, *exploring unmerged tokens*. In this work, we applied this approach by training on multiple different tokenizations for each edge. While this led to minor improvements, it also significantly increased computational costs. Other potential strategies include randomly selecting the tokenization during each epoch or incorporating different tokenizations directly into the embedding process. An example of the latter would be averaging the embeddings of all possible tokenizations. Third, *foregoing subword tokenization* entirely. Instead, using character-level tokenization or even the raw bytes of the labels would drastically reduce the vocabulary size, thereby lowering the proportion of tokens not encountered by BYTE. Notably, as long as a character is found in the training data, it will receive some attention during training. In contrast, subword tokenization may skip a token if it is part of another token. This approach aligns with recent advances in tokenless transformers, which operate directly on byte or character sequences and could potentially replace the embedding layer proposed in BYTE. While promising, especially for addressing the lack of token exploration, such models are generally more computationally expensive due to longer input sequences.

9. Acknowledgements

The author gratefully acknowledges the computing time provided on the high-performance computer HoreKa by the National High-Performance Computing Center at KIT (NHR@KIT). This center is jointly supported by the Federal Ministry of Education and Research and the Ministry of Science, Research and the Arts of Baden-Württemberg, as part of the National High-Performance Computing (NHR) joint funding program (https://www.nhr-verein.de/en/our-partners). HoreKa is partly funded by the German Research Foundation (DFG).

References

[1] S. Ji, S. Pan, E. Cambria, P. Marttinen, P.S. Yu, "A survey on knowledge graphs: Representation, acquisition, and applications", *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, no. 2, pp. 494–514, 2022. DOI: `10.1109/TNNLS.2021.3070843` (cit. on p. 1).

[2] J. Cao, J. Fang, Z. Meng, S. Liang, "Knowledge graph embedding: A survey from the perspective of representation spaces", *ACM Computing Surveys*, vol. 56, no. 6, pp. 1–42, 2024 (cit. on p. 1).

[3] K. K. Teru, E. Denis, W. L. Hamilton, "Inductive relation prediction by subgraph reasoning.", *arXiv: Learning*, 2020 (cit. on pp. 1, 3, 7, 14).

[4] M. Galkin, X. Yuan, H. Mostafa, J. Tang, Z. Zhu, "Towards foundation models for knowledge graph reasoning", in *The Twelfth International Conference on Learning Representations*, 2024. [Online]. Available: `https://openreview.net/forum?id=jVEoydFOl9` (cit. on pp. 1, 3, 7, 19).

[5] W. Hamilton, Z. Ying, J. Leskovec, "Inductive representation learning on large graphs", *Advances in neural information processing systems*, vol. 30, 2017 (cit. on pp. 1, 6).

[6] D. Daza, M. Cochez, P. Groth, "Inductive entity representations from text via link prediction", in *Proceedings of the Web Conference 2021*, 2021, pp. 798–808 (cit. on pp. 1, 6).

[7] A. Sadeghian, M. Armandpour, P. Ding, D. Z. Wang, "Drum: End-to-end differentiable rule mining on knowledge graphs", *Advances in Neural Information Processing Systems*, vol. 32, 2019 (cit. on pp. 1, 6).

[8] H. R. Qian Huang, J. Leskovec, "Few-shot relational reasoning via connection subgraph pretraining", in *Neural Information Processing Systems*, 2022 (cit. on pp. 1, 7).

[9] M. Galkin, E. Denis, J. Wu, W. L. Hamilton, "Nodepiece: Compositional and parameter-efficient representations of large knowledge graphs", in *International Conference on Learning Representations*, 2022. [Online]. Available: `https://openreview.net/forum?id=xMJWUKJnFSw` (cit. on pp. 1, 7).

[10] C. Demir, N. Kouagou, A. Sharma, A.-C. N. Ngomo, "Inference over unseen entities, relations and literals on knowledge graphs", *arXiv preprint arXiv:2410.06742*, 2024 (cit. on pp. 1, 6, 8, 16, 19, 21).

[11] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, *et al.*, "Language models are unsupervised multitask learners", *OpenAI blog*, vol. 1, no. 8, p. 9, 2019 (cit. on pp. 1, 4, 7).

[12] F. Manola, E. Miller, B. McBride, *et al.*, "Rdf primer", *W3C recommendation*, vol. 10, no. 1-107, p. 6, 2004 (cit. on p. 3).

[13] S. J. Mielke, Z. Alyafeai, E. Salesky, C. Raffel, M. Dey, M. Gallé, A. Raja, C. Si, W. Y. Lee, B. Sagot, S. Tan, "Between words and characters: A brief history of open-vocabulary modeling and tokenization in NLP", *CoRR*, vol. abs/2112.10508, 2021. arXiv: `2112.10508`. [Online]. Available: `https://arxiv.org/abs/2112.10508` (cit. on p. 4).

[14] C. W. Schmidt, V. Reddy, H. Zhang, A. Alameddine, O. Uzan, Y. Pinter, C. Tanner, "Tokenization is more than compression", *CoRR*, vol. abs/2402.18376, 2024. DOI: `10.48550/ARXIV.2402.18376`. arXiv: `2402.18376`. [Online]. Available: `https://doi.org/10.48550/arXiv.2402.18376` (cit. on p. 4).

[15] P. Gage, "A new algorithm for data compression", *C Users J.*, vol. 12, no. 2, pp. 23–38, Feb. 1994, ISSN: 0898-9788 (cit. on p. 4).

[16] R. Sennrich, B. Haddow, A. Birch, "Neural machine translation of rare words with subword units", in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*, The Association for Computer Linguistics, 2016. DOI: `10.18653/V1/P16-1162`. [Online]. Available: `https://doi.org/10.18653/v1/p16-1162` (cit. on p. 4).

[17] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation", *arXiv preprint arXiv:1609.08144*, 2016 (cit. on p. 4).

[18] T. Kudo, "Subword regularization: Improving neural network translation models with multiple subword candidates", in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, I. Gurevych, Y. Miyao, Eds., Melbourne, Australia: Association for Computational Linguistics, Jul. 2018, pp. 66–75. DOI: `10.18653/v1/P18-1007`. [Online]. Available: `https://aclanthology.org/P18-1007/` (cit. on p. 4).

[19] I. D. Mienye, T. G. Swart, G. Obaido, "Recurrent neural networks: A comprehensive review of architectures, variants, and applications", *Information*, vol. 15, no. 9, 2024, ISSN: 2078-2489. [Online]. Available: `https://www.mdpi.com/2078-2489/15/9/517` (cit. on p. 5).

[20] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, I. Polosukhin, "Attention is all you need", *Advances in neural information processing systems*, vol. 30, 2017 (cit. on p. 5).

[21] J. L. Elman, "Finding structure in time", *Cognitive science*, vol. 14, no. 2, pp. 179–211, 1990 (cit. on p. 5).

[22] K. Cho, B. van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation", in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Association for Computational Linguistics, 2014, pp. 1724–1734. DOI: 10.3115/v1/D14-1179. [Online]. Available: https://arxiv.org/abs/1406.1078 (cit. on p. 5).

[23] S. Hochreiter, J. Schmidhuber, "Long short-term memory", *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997 (cit. on p. 5).

[24] A. Bordes, N. Usunier, A. Garcia-Duran, J. Weston, O. Yakhnenko, "Translating embeddings for modeling multi-relational data", *Advances in neural information processing systems*, vol. 26, 2013 (cit. on pp. 6, 14).

[25] B. Yang, W. Yih, X. He, J. Gao, L. Deng, "Embedding entities and relations for learning and inference in knowledge bases", in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio, Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1412.6575 (cit. on p. 6).

[26] T. Trouillon, J. Welbl, S. Riedel, É. Gaussier, G. Bouchard, "Complex embeddings for simple link prediction", in *International conference on machine learning*, PMLR, 2016, pp. 2071–2080 (cit. on p. 6).

[27] C. Demir, A.-C. Ngonga Ngomo, "Clifford embeddings–a generalized approach for embedding in normed algebras", in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, Springer, 2023, pp. 567–582 (cit. on p. 6).

[28] G. A. Gesese, H. Sack, M. Alam, "Raild: Towards leveraging relation features for inductive link prediction in knowledge graphs", in *Proceedings of the 11th International Joint Conference on Knowledge Graphs*, ser. IJCKG '22, Hangzhou, China: Association for Computing Machinery, 2023, pp. 82–90, ISBN: 9781450399876. DOI: 10.1145/3579051.3579066. [Online]. Available: https://doi.org/10.1145/3579051.3579066 (cit. on p. 6).

[29] C. Meilicke, M. W. Chekol, P. Betz, M. Fink, H. Stuckeschmidt, "Anytime bottom-up rule learning for large-scale knowledge graph completion", *The VLDB Journal*, vol. 33, no. 1, pp. 131–161, 2024 (cit. on p. 6).

[30] T. Shen, F. Zhang, J. Cheng, "A comprehensive overview of knowledge graph completion", *Knowledge-Based Systems*, vol. 255, p. 109 597, 2022 (cit. on p. 14).

[31] W. Xiong, T. Hoang, W. Y. Wang, "DeepPath: A reinforcement learning method for knowledge graph reasoning", in *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, M. Palmer, R. Hwa, S. Riedel, Eds., Copenhagen, Denmark: Association for Computational Linguistics, Sep. 2017, pp. 564–573. DOI: 10.18653/v1/D17-1060. [Online]. Available: https://aclanthology.org/D17-1060 (cit. on p. 14).

[32] K. Toutanova, D. Chen, “Observed versus latent features for knowledge base and text inference”, in *Proceedings of the 3rd workshop on continuous vector space models and their compositionality*, 2015, pp. 57–66 (cit. on p. 14).

[33] J. Lee, C. Chung, J. J. Whang, “Ingram: Inductive knowledge graph embedding via relation graphs”, in *International Conference on Machine Learning*, PMLR, 2023, pp. 18 796–18 809 (cit. on p. 14).

[34] I. Balazevic, C. Allen, T. Hospedales, “Multi-relational poincaré graph embeddings”, *Advances in neural information processing systems*, vol. 32, 2019 (cit. on pp. 14, 19, 21).

[35] H. Shomer, J. Revolinsky, J. Tang, “Towards better benchmark datasets for inductive knowledge graph completion”, in *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2025 (cit. on p. 21).

A. Datasets

Table 11: **Dataset statistics.** The table shows the distribution of entities V, relation types R, and triples E. It also shows the mean minimum Levenshtein distance, which measures the differences between seen and unseen elements.

Dataset	$\|V_{\text{train}}\|$	$\|R_{\text{train}}\|$	$\|E_{\text{train}}\|$	$\|V_{\text{test}}\|$	$\|R_{\text{test}}\|$	$\|E_{\text{test}}\|$	$\|V_{\text{unseen}}\|$	$\|R_{\text{unseen}}\|$	$\|E_{\text{unseen}}\|$	Mean min. Levenshtein
FB V1	1592	180	4245	301	68	205	300	0	205	5.501
FB V2	2604	200	9739	562	107	478	560	0	478	5.593
FB V3	3648	215	17986	980	128	865	970	0	865	5.396
FB V4	4680	219	27203	1425	166	1424	1414	0	1424	5.596
NELL V1	3103	14	4687	84	7	100	84	0	100	1.659
NELL V2	2564	88	8219	478	54	476	478	0	476	5.530
NELL V3	4647	142	16393	798	87	809	796	0	809	5.530
NELL V4	2092	76	7546	630	47	731	629	0	731	7.388
WN V1	2758	9	5410	288	6	188	288	0	188	3.061
WN V2	6988	10	15262	710	9	441	710	0	441	2.650
WN V3	12142	11	25901	977	10	605	976	0	605	3.576
WN V4	3885	9	7940	2273	9	1429	2273	0	1429	2.907
NL-0	1814	134	7796	593	82	763	593	0	763	5.170
NL-100	1258	55	7832	591	38	793	591	38	793	7.824
NL-25	4396	106	17578	598	75	744	598	35	744	4.851
NL-50	4396	106	17578	673	81	859	673	41	859	5.227
NL-75	2607	96	11058	475	62	607	475	38	607	6.279
NELL-h25	64634	172	122618	10245	172	9187	2831	0	2459	1.435
NELL-h50	32277	86	72767	5582	86	5393	1196	0	1180	1.016
NELL-h75	26219	57	59135	4600	57	4389	978	0	950	1.051
NELL-h100	21010	43	50314	3833	43	3746	755	0	738	0.976

B. Full results

This section contains the list of hyperparameter configurations and the MRR and H@10 results achieved by experiments with said configuration. Additionally, it includes the results of experiments conducted initially with inference graphs used as training data.

The list of configurations is split across Tables 12, 14, 16 and 18. Each table contains all parameters for which the parameter search was performed, and the Configuration column that assigns a shorthand name to each configuration, based on the hyperparameters that they represent. For example, "2 True" refers to the best configuration found for multitoken = 2 with multiloss enabled. Tables 13, 15, 17 and 19 then report each configuration's mean MRR and H@10 scores, as well as their corresponding standard deviations.

Regarding the experiments trained on the inference graph, all datasets labeled the inference graph files as "train.txt," which we initially misinterpreted as the actual training data. Although these experiments were limited, being brief exploratory tests and not conducted on established benchmarks, they still provide valuable insights into BYTE. First, vocabularies fitted to knowledge graphs generally performed better than GPT-2 in the transductive setting. Second, vocabulary effectiveness is dataset-dependent.

Table 12: **Hyperparameter configurations for BYTE with different vocabularies.** Each row represents one hyperparameter configuration on KG_pool = Configuration.

Config.	training_KG	KG_pool	corpus_input	tie_breaker	vocab_size	bpe_with_RNN	model	embedding_dim	lr
KG-custom	FB-v3-ind	KG-custom	E	descending_size	1.00	Linear	Keci	64	0.01
KG-finetuned	FB-v3-ind	KG-finetuned	VandR	ascending_size	0.50	Linear	Keci	32	0.01
KG-pretrained	FB-v3-ind	KG-pretrained	VandR	descending_size	2.00	Linear	ComplEx	64	0.01
original_BytE	FB-v3-ind	original_BytE	None	None	1.00	Linear	Keci	64	0.01
KG-custom	NELL-995-h25	KG-custom	randomWalks	descending_size	2.00	Linear	ComplEx	16	0.01
KG-finetuned	NELL-995-h25	KG-finetuned	VandR	ascending_size	2.00	Linear	ComplEx	16	0.01
KG-pretrained	NELL-995-h25	KG-pretrained	VandR	ascending_size	2.00	Linear	ComplEx	64	0.01
original_BytE	NELL-995-h25	original_BytE	None	None	1.00	Linear	ComplEx	32	0.01
KG-custom	NELL-995-h75	KG-custom	E	descending_size	1.00	Linear	ComplEx	32	0.01
KG-finetuned	NELL-995-h75	KG-finetuned	VandR	descending_size	2.00	Linear	ComplEx	32	0.01
KG-pretrained	NELL-995-h75	KG-pretrained	VandR	ascending_size	0.50	Linear	ComplEx	64	0.01
original_BytE	NELL-995-h75	original_BytE	None	None	1.00	Linear	ComplEx	32	0.01
KG-custom	NL-25	KG-custom	VandR	descending_size	0.50	Linear	Keci	16	0.01
KG-finetuned	NL-25	KG-finetuned	VandR	descending_size	0.50	Linear	Keci	16	0.01
KG-pretrained	NL-25	KG-pretrained	VandR	descending_size	0.50	Linear	Keci	16	0.01
original_BytE	NL-25	original_BytE	None	None	1.00	Linear	ComplEx	64	0.01
KG-custom	NL-v1-ind	KG-custom	E	descending_size	2.00	Linear	Keci	32	0.10
KG-finetuned	NL-v1-ind	KG-finetuned	VandR	ascending_size	0.50	Linear	ComplEx	32	0.10
KG-pretrained	NL-v1-ind	KG-pretrained	VandR	descending_size	0.50	Linear	Keci	32	0.10
original_BytE	NL-v1-ind	original_BytE	None	None	1.00	Linear	ComplEx	64	0.01
KG-custom	NL-v4-ind	KG-custom	randomWalks	descending_size	1.00	Linear	ComplEx	32	0.01
KG-finetuned	NL-v4-ind	KG-finetuned	VandR	ascending_size	1.00	Linear	ComplEx	64	0.01
KG-pretrained	NL-v4-ind	KG-pretrained	VandR	descending_size	1.00	Linear	ComplEx	32	0.01
original_BytE	NL-v4-ind	original_BytE	None	None	1.00	Linear	ComplEx	64	0.01
KG-custom	wn-v2-ind	KG-custom	randomWalks	descending_size	0.50	Linear	Keci	16	0.01
KG-finetuned	wn-v2-ind	KG-finetuned	VandR	ascending_size	0.50	Linear	ComplEx	32	0.01
KG-pretrained	wn-v2-ind	KG-pretrained	VandR	descending_size	0.50	Linear	ComplEx	32	0.01
original_BytE	wn-v2-ind	original_BytE	None	None	1.00	Linear	ComplEx	32	0.01
KG-custom	wn-v4-ind	KG-custom	E	ascending_size	0.50	Linear	ComplEx	64	0.01
KG-finetuned	wn-v4-ind	KG-finetuned	VandR	descending_size	0.50	Linear	Keci	64	0.01
KG-pretrained	wn-v4-ind	KG-pretrained	VandR	ascending_size	0.50	Linear	ComplEx	32	0.10
original_BytE	wn-v4-ind	original_BytE	None	None	1.00	Linear	Keci	64	0.10

Table 13: **Results of the vocabulary evaluation**, showing mean values and standard deviation for the two metrics mean reciprocal rank (MRR) and H@10.

Config.	NELL-h75		NELL-h25	
	Test H@10	Test MRR	Test H@10	Test MRR
KG-custom	36.19% ± 08.31%	23.00% ± 05.61%	08.71% ± 02.17%	04.64% ± 01.34%
KG-finetuned	33.46% ± 14.90%	21.50% ± 10.60%	25.41% ± 10.77%	16.38% ± 07.60%
KG-pretrained	27.20% ± 08.66%	16.95% ± 06.16%	03.53% ± —	01.65% ± —
GPT-2	42.59% ± 00.57%	27.82% ± 00.32%	11.27% ± —	06.49% ± —

Config.	NELL V1		WN V2	
	Test H@10	Test MRR	Test H@10	Test MRR
KG-custom	0.00% ± 0.00%	0.06% ± 0.01%	0.59% ± 0.64%	0.39% ± 0.40%
KG-finetuned	0.80% ± 1.25%	0.51% ± 0.29%	0.20% ± 0.15%	0.12% ± 0.05%
KG-pretrained	1.80% ± 2.56%	0.45% ± 0.46%	1.79% ± 2.65%	0.90% ± 1.18%
GPT-2	4.20% ± 3.17%	1.40% ± 0.77%	0.86% ± 0.68%	0.38% ± 0.22%

Config.	WN V4		NL-25	
	Test H@10	Test MRR	Test H@10	Test MRR
KG-custom	0.05% ± 0.09%	0.06% ± 0.03%	1.98% ± 0.25%	0.73% ± 0.16%
KG-finetuned	0.32% ± 0.37%	0.23% ± 0.20%	2.72% ± 0.51%	1.04% ± 0.27%
KG-pretrained	0.15% ± 0.10%	0.12% ± 0.04%	0.95% ± 0.63%	0.46% ± 0.22%
GPT-2	0.16% ± 0.11%	0.10% ± 0.06%	1.06% ± 0.82%	0.40% ± 0.14%

Config.	FB V3		NELL V4	
	Test H@10	Test MRR	Test H@10	Test MRR
KG-custom	0.18% ± 0.18%	0.15% ± 0.07%	0.62% ± 0.57%	0.33% ± 0.20%
KG-finetuned	0.25% ± 0.10%	0.19% ± 0.03%	0.15% ± 0.09%	0.17% ± 0.02%
KG-pretrained	0.24% ± 0.06%	0.22% ± 0.02%	0.30% ± 0.17%	0.22% ± 0.08%
GPT-2	0.60% ± 0.17%	0.28% ± 0.09%	0.42% ± 0.18%	0.28% ± 0.13%

Table 14: **Hyperparameter configurations for BYTE with RNNs for the matrix reduction.** Each row represents one hyperparameter configuration on bpe_with_RNN = Configuration.

Config.	training_KG	KG_pool	corpus_input	tie_breaker	vocab_size	bpe_with_RNN	model	embedding_dim	lr
GRU	FB-v3-ind	original_BytE	None	None	1.00	GRU	ComplEx	64	0.10
LSTM	FB-v3-ind	KG-pretrained	VandR	descending_size	2.00	LSTM	ComplEx	64	0.10
RNN	FB-v3-ind	original_BytE	None	None	1.00	RNN	Keci	64	0.01
GRU	NELL-995-h25	original_BytE	None	None	1.00	GRU	ComplEx	64	0.01
GRU	NELL-995-h25	KG-custom	randomWalks	descending_size	2.00	GRU	ComplEx	64	0.01
RNN	NELL-995-h25	original_BytE	None	None	1.00	RNN	ComplEx	64	0.01
GRU	NELL-995-h75	KG-custom	E	descending_size	1.00	GRU	ComplEx	64	0.01
LSTM	NELL-995-h75	KG-custom	E	descending_size	1.00	LSTM	ComplEx	64	0.01
RNN	NELL-995-h75	KG-custom	E	descending_size	1.00	RNN	ComplEx	64	0.01
GRU	NL-25	original_BytE	None	None	1.00	GRU	ComplEx	32	0.10
GRU	NL-25	KG-finetuned	VandR	descending_size	0.50	GRU	Keci	64	0.10
LSTM	NL-25	original_BytE	None	None	1.00	LSTM	Keci	32	0.01
LSTM	NL-25	KG-finetuned	VandR	descending_size	0.50	LSTM	ComplEx	16	0.01
RNN	NL-25	original_BytE	None	None	1.00	RNN	Keci	16	0.01
RNN	NL-25	KG-finetuned	VandR	descending_size	0.50	RNN	Keci	32	0.01
GRU	NL-v1-ind	original_BytE	None	None	1.00	GRU	Keci	64	0.10
LSTM	NL-v1-ind	KG-pretrained	VandR	descending_size	0.50	LSTM	ComplEx	16	0.10
LSTM	NL-v1-ind	original_BytE	None	None	1.00	LSTM	Keci	16	0.01
RNN	NL-v1-ind	KG-pretrained	VandR	descending_size	0.50	RNN	ComplEx	32	0.10
GRU	NL-v4-ind	original_BytE	None	None	1.00	GRU	Keci	64	0.10
GRU	NL-v4-ind	KG-custom	randomWalks	descending_size	1.00	GRU	Keci	64	0.10
LSTM	NL-v4-ind	KG-custom	randomWalks	descending_size	1.00	LSTM	Keci	32	0.10
LSTM	NL-v4-ind	original_BytE	None	None	1.00	LSTM	ComplEx	64	0.01
RNN	NL-v4-ind	KG-custom	randomWalks	descending_size	1.00	RNN	Keci	32	0.10
RNN	NL-v4-ind	original_BytE	None	None	1.00	RNN	Keci	32	0.10
GRU	wn-v2-ind	KG-finetuned	VandR	ascending_size	0.50	GRU	Keci	64	0.10
LSTM	wn-v2-ind	KG-finetuned	VandR	ascending_size	0.50	LSTM	Keci	16	0.10
RNN	wn-v2-ind	KG-finetuned	VandR	ascending_size	0.50	RNN	ComplEx	64	0.10
GRU	wn-v4-ind	KG-finetuned	VandR	descending_size	0.50	GRU	ComplEx	16	0.01
LSTM	wn-v4-ind	original_BytE	None	None	1.00	LSTM	ComplEx	32	0.01
LSTM	wn-v4-ind	KG-finetuned	VandR	descending_size	0.50	LSTM	ComplEx	16	0.01
RNN	wn-v4-ind	original_BytE	None	None	1.00	RNN	Keci	16	0.10
RNN	wn-v4-ind	KG-finetuned	VandR	descending_size	0.50	RNN	ComplEx	16	0.01

Table 15: **Results of the matrix reduction evaluation**, showing mean values and standard deviation for the two metrics mean reciprocal rank (MRR) and H@10.

Config.	NELL-h75		NELL-h25	
	Test H@10	Test MRR	Test H@10	Test MRR
GRU	42.89% ± 3.32%	28.02% ± 2.18%	37.17% ± 3.42%	25.04% ± 2.22%
LSTM	45.41% ± 0.39%	29.82% ± 0.25%	—	—
RNN	42.92% ± 0.72%	27.90% ± 0.49%	16.43% ± —	9.81% ± —

Config.	NELL V1		WN V2	
	Test H@10	Test MRR	Test H@10	Test MRR
GRU	0.20% ± 0.27%	0.54% ± 0.33%	3.97% ± 2.37%	1.65% ± 0.90%
LSTM	0.00% ± 0.00%	0.12% ± 0.05%	0.00% ± 0.00%	0.04% ± 0.03%
RNN	0.00% ± 0.00%	0.21% ± 0.30%	0.00% ± 0.00%	0.03% ± 0.00%

Config.	WN V4		NL-25	
	Test H@10	Test MRR	Test H@10	Test MRR
GRU	0.02% ± 0.03%	0.06% ± 0.02%	0.02% ± 0.03%	0.10% ± 0.05%
LSTM	0.00% ± 0.00%	0.04% ± 0.02%	0.00% ± 0.00%	0.07% ± 0.02%
RNN	0.01% ± 0.02%	0.05% ± 0.02%	0.01% ± 0.04%	0.08% ± 0.02%

Config.	FB V3		NELL V4	
	Test H@10	Test MRR	Test H@10	Test MRR
GRU	0.37% ± 0.40%	0.26% ± 0.14%	0.24% ± 0.12%	0.19% ± 0.09%
LSTM	0.16% ± 0.24%	0.19% ± 0.06%	0.06% ± 0.17%	0.12% ± 0.08%
RNN	0.30% ± 0.32%	0.21% ± 0.09%	0.00% ± 0.00%	0.09% ± 0.02%

Table 16: **Hyperparameter configurations for BYTE with truncation.** Each row represents one hyperparameter configuration, where Config. represents normalization with a certain strategy up to a normalization length of n%.

Config.	training_KG	KG_pool	corpus_input	tie_breaker	vocab_size	bpe_with_RNN	model	embedding_dim	lr
asc_rank@90	FB-v3-ind	original_BytE	None	None	1.00	Linear	Keci	32	0.01
asc_size@90	FB-v3-ind	KG-pretrained	VandR	descending_size	2.00	RNN	Keci	16	0.10
desc_rank@90	FB-v3-ind	original_BytE	None	None	1.00	RNN	Keci	64	0.01
desc_size@95	FB-v3-ind	KG-pretrained	VandR	descending_size	2.00	RNN	ComplEx	16	0.10
last_first@95	FB-v3-ind	original_BytE	None	None	1.00	RNN	Keci	64	0.01
asc_size@90	NELL-995-h75	KG-custom	E	descending_size	2.00	Linear	Keci	64	0.01
desc_rank@90	NELL-995-h75	KG-custom	E	descending_size	2.00	Linear	Keci	64	0.01
desc_size@90	NELL-995-h75	KG-custom	E	descending_size	2.00	Linear	ComplEx	64	0.01
asc_rank@95	NELL-995-h75	KG-custom	E	descending_size	2.00	Linear	Keci	64	0.01
asc_size@95	NELL-995-h75	KG-custom	E	descending_size	2.00	Linear	Keci	64	0.01
last_first@95	NELL-995-h75	KG-custom	E	descending_size	2.00	Linear	ComplEx	64	0.01
desc_rank@90	NL-25	KG-finetuned	VandR	descending_size	0.50	Linear	Keci	64	0.01
asc_rank@95	NL-25	KG-finetuned	VandR	descending_size	0.50	Linear	Keci	16	0.01
asc_size@95	NL-25	KG-finetuned	VandR	descending_size	0.50	Linear	ComplEx	64	0.01
desc_size@95	NL-25	KG-finetuned	VandR	descending_size	0.50	Linear	Keci	16	0.01
last_first@95	NL-25	KG-finetuned	VandR	descending_size	0.50	Linear	Keci	16	0.01
desc_rank@90	NL-v1-ind	original_BytE	None	None	1.00	Linear	Keci	64	0.01
desc_size@90	NL-v1-ind	original_BytE	None	None	1.00	Linear	ComplEx	64	0.01
asc_rank@95	NL-v1-ind	original_BytE	None	None	1.00	Linear	ComplEx	32	0.10
asc_size@95	NL-v1-ind	original_BytE	None	None	1.00	Linear	ComplEx	32	0.01
desc_size@95	NL-v1-ind	original_BytE	None	None	1.00	Linear	ComplEx	32	0.01
last_first@95	NL-v1-ind	original_BytE	None	None	1.00	Linear	ComplEx	64	0.01
asc_size@90	NL-v4-ind	KG-custom	randomWalks	descending_size	1.00	Linear	Keci	64	0.01
desc_rank@90	NL-v4-ind	original_BytE	None	None	1.00	GRU	ComplEx	64	0.10
last_first@90	NL-v4-ind	KG-custom	randomWalks	descending_size	1.00	Linear	Keci	64	0.01
asc_rank@95	NL-v4-ind	original_BytE	None	None	1.00	Linear	ComplEx	64	0.01
asc_size@95	NL-v4-ind	original_BytE	None	None	1.00	Linear	ComplEx	64	0.01
desc_size@95	NL-v4-ind	original_BytE	None	None	1.00	Linear	ComplEx	64	0.10
asc_rank@90	wn-v2-ind	original_BytE	None	None	1.00	Linear	Keci	32	0.01
desc_size@90	wn-v2-ind	original_BytE	None	None	1.00	Linear	Keci	32	0.01
asc_size@95	wn-v2-ind	original_BytE	None	None	1.00	Linear	Keci	32	0.01
desc_rank@95	wn-v2-ind	original_BytE	None	None	1.00	Linear	Keci	32	0.01
desc_size@95	wn-v2-ind	original_BytE	None	None	1.00	Linear	Keci	32	0.01
last_first@95	wn-v2-ind	original_BytE	None	None	1.00	Linear	Keci	32	0.01
asc_rank@90	wn-v4-ind	KG-finetuned	VandR	descending_size	0.50	Linear	ComplEx	64	0.01
desc_size@90	wn-v4-ind	KG-finetuned	VandR	descending_size	0.50	Linear	ComplEx	64	0.01
last_first@90	wn-v4-ind	original_BytE	None	None	1.00	Linear	Keci	32	0.01
asc_size@95	wn-v4-ind	original_BytE	None	None	1.00	Linear	Keci	32	0.01
desc_rank@95	wn-v4-ind	KG-finetuned	VandR	descending_size	0.50	Linear	ComplEx	64	0.01
desc_size@95	wn-v4-ind	KG-finetuned	VandR	descending_size	0.50	Linear	ComplEx	64	0.01

Table 17: **Results of the Truncation evaluation**, showing mean values and standard deviation for the two metrics mean reciprocal rank (MRR) and H@10.

	NELL-h75	
	Test H@10	Test MRR
asc_size@90	40.48% ± 00.66%	26.13% ± 0.59%
desc_rank@90	40.63% ± 01.12%	26.51% ± 0.92%
desc_size@90	37.61% ± 00.70%	23.47% ± 0.55%
asc_rank@95	41.13% ± 00.63%	26.75% ± 0.58%
asc_size@95	40.48% ± 00.66%	26.13% ± 0.59%
last_first@95	32.39% ± 10.97%	19.88% ± 7.33%

	NELL V1	
	Test H@10	Test MRR
desc_rank@90	0.00% ± 0.00%	0.25% ± 0.07%
desc_size@90	0.30% ± 0.45%	0.32% ± 0.27%
asc_rank@95	0.00% ± 0.00%	0.19% ± 0.14%
asc_size@95	0.20% ± 0.45%	0.23% ± 0.10%
desc_size@95	0.30% ± 0.45%	0.30% ± 0.12%
last_first@95	0.00% ± 0.00%	0.23% ± 0.11%

	WN V2	
	Test H@10	Test MRR
asc_rank@90	1.45% ± 1.33%	0.73% ± 0.75%
desc_size@90	1.61% ± 1.55%	0.76% ± 0.68%
asc_size@95	0.73% ± 0.60%	0.35% ± 0.29%
desc_rank@95	0.77% ± 0.50%	0.47% ± 0.39%
desc_size@95	1.61% ± 1.55%	0.76% ± 0.68%
last_first@95	1.56% ± 1.52%	0.73% ± 0.71%

	WN V4	
	Test H@10	Test MRR
asc_rank@90	0.27% ± 0.40%	0.21% ± 0.23%
desc_size@90	0.31% ± 0.27%	0.20% ± 0.14%
last_first@90	0.64% ± 0.82%	0.31% ± 0.34%
asc_size@95	0.58% ± 0.87%	0.30% ± 0.37%
desc_rank@95	1.90% ± 2.41%	1.01% ± 1.19%
desc_size@95	0.31% ± 0.27%	0.20% ± 0.14%

	NL-25	
	Test H@10	Test MRR
desc_rank@90	2.28% ± 1.69%	0.78% ± 0.31%
asc_rank@95	2.39% ± 0.13%	1.12% ± 0.15%
asc_size@95	0.90% ± 0.21%	0.44% ± 0.09%
desc_size@95	3.20% ± 1.21%	1.17% ± 0.36%
last_first@95	2.55% ± 0.37%	1.13% ± 0.22%

	FB V3	
	Test H@10	Test MRR
asc_rank@90	0.07% ± 0.09%	0.09% ± 0.03%
asc_size@90	0.18% ± 0.26%	0.16% ± 0.12%
desc_rank@90	0.25% ± 0.26%	0.22% ± 0.07%
desc_size@95	0.61% ± 0.74%	0.34% ± 0.31%
last_first@95	0.23% ± 0.11%	0.21% ± 0.05%

	NELL V4	
	Test H@10	Test MRR
asc_size@90	0.49% ± 0.30%	0.33% ± 0.15%
desc_rank@90	0.37% ± 0.21%	0.19% ± 0.08%
last_first@90	0.66% ± 0.14%	0.41% ± 0.07%
asc_rank@95	0.16% ± 0.06%	0.14% ± 0.04%
asc_size@95	0.16% ± 0.09%	0.17% ± 0.07%
desc_size@95	0.23% ± 0.23%	0.13% ± 0.06%

Table 18: **Hyperparameter configurations for BYTE with multitoken.** Each row represents one hyperparameter configuration, where Config. = n True stands for n tokenizations per edge with multiloss enabled.

Config.	training_KG	KG_pool	corpus_input	tie_breaker	vocab_size	bpe_with_RNN	model	embedding_dim	lr
2 False	FB-v3-ind	KG-pretrained	VandR	descending_size	2.00	Linear	Keci	64	0.01
2 True	FB-v3-ind	KG-pretrained	VandR	descending_size	2.00	Linear	Keci	64	0.01
3 False	FB-v3-ind	original_BytE	None	None	1.00	Linear	Keci	16	0.10
3 True	FB-v3-ind	original_BytE	None	None	1.00	Linear	Keci	16	0.10
4 False	FB-v3-ind	original_BytE	None	None	1.00	RNN	Keci	64	0.01
4 True	FB-v3-ind	original_BytE	None	None	1.00	RNN	Keci	64	0.01
2 False	NELL-995-h75	KG-custom	E	descending_size	2.00	Linear	Keci	32	0.01
2 True	NELL-995-h75	KG-custom	E	descending_size	2.00	Linear	Keci	32	0.01
3 False	NELL-995-h75	KG-custom	E	descending_size	2.00	Linear	Keci	32	0.01
3 True	NELL-995-h75	KG-custom	E	descending_size	2.00	Linear	Keci	32	0.01
4 False	NELL-995-h75	KG-custom	E	descending_size	2.00	Linear	ComplEx	16	0.01
4 True	NELL-995-h75	KG-custom	E	descending_size	2.00	Linear	ComplEx	16	0.01
2 False	NL-25	KG-finetuned	VandR	descending_size	0.50	Linear	ComplEx	64	0.01
2 True	NL-25	KG-finetuned	VandR	descending_size	0.50	Linear	ComplEx	64	0.01
3 False	NL-25	KG-finetuned	VandR	descending_size	0.50	Linear	ComplEx	16	0.01
3 True	NL-25	KG-finetuned	VandR	descending_size	0.50	Linear	ComplEx	16	0.01
4 False	NL-25	KG-finetuned	VandR	descending_size	0.50	Linear	ComplEx	16	0.01
4 True	NL-25	KG-finetuned	VandR	descending_size	0.50	Linear	ComplEx	16	0.01
2 False	NL-v1-ind	original_BytE	None	None	1.00	Linear	Keci	64	0.10
2 True	NL-v1-ind	original_BytE	None	None	1.00	Linear	Keci	64	0.10
3 False	NL-v1-ind	original_BytE	None	None	1.00	Linear	ComplEx	16	0.10
3 True	NL-v1-ind	original_BytE	None	None	1.00	Linear	ComplEx	16	0.10
4 False	NL-v1-ind	original_BytE	None	None	1.00	Linear	ComplEx	16	0.10
4 True	NL-v1-ind	original_BytE	None	None	1.00	Linear	ComplEx	16	0.10
2 False	NL-v4-ind	original_BytE	None	None	1.00	Linear	Keci	64	0.01
2 True	NL-v4-ind	original_BytE	None	None	1.00	Linear	Keci	64	0.01
3 False	NL-v4-ind	original_BytE	None	None	1.00	GRU	Keci	64	0.10
3 True	NL-v4-ind	original_BytE	None	None	1.00	GRU	Keci	64	0.10
4 False	NL-v4-ind	KG-custom	randomWalks	descending_size	1.00	RNN	ComplEx	64	0.01
4 True	NL-v4-ind	KG-custom	randomWalks	descending_size	1.00	RNN	ComplEx	64	0.01
2 False	wn-v2-ind	KG-finetuned	VandR	ascending_size	0.50	Linear	Keci	16	0.01
2 True	wn-v2-ind	KG-finetuned	VandR	ascending_size	0.50	Linear	Keci	16	0.01
3 False	wn-v2-ind	KG-finetuned	VandR	ascending_size	0.50	Linear	Keci	32	0.01
3 True	wn-v2-ind	KG-finetuned	VandR	ascending_size	0.50	Linear	Keci	32	0.01
4 False	wn-v2-ind	KG-finetuned	VandR	ascending_size	0.50	Linear	Keci	32	0.01
4 True	wn-v2-ind	KG-finetuned	VandR	ascending_size	0.50	Linear	Keci	32	0.01
2 False	wn-v4-ind	KG-finetuned	VandR	descending_size	0.50	Linear	ComplEx	64	0.10
2 True	wn-v4-ind	KG-finetuned	VandR	descending_size	0.50	Linear	ComplEx	64	0.10
3 False	wn-v4-ind	KG-finetuned	VandR	descending_size	0.50	Linear	Keci	32	0.10
3 True	wn-v4-ind	KG-finetuned	VandR	descending_size	0.50	Linear	Keci	32	0.10
4 False	wn-v4-ind	KG-finetuned	VandR	descending_size	0.50	Linear	ComplEx	64	0.10
4 True	wn-v4-ind	KG-finetuned	VandR	descending_size	0.50	Linear	ComplEx	64	0.10

Table 19: **Results of the multitoken evaluation**, showing mean values and standard deviation for the two metrics mean reciprocal rank (MRR) and H@10.

Config.	NELL-h75		NELL V1	
	Test H@10	Test MRR	Test H@10	Test MRR
2 False	28.95% ± 12.63%	17.95% ± 8.77%	0.00% ± 0.00%	0.09% ± 0.04%
2 True	2.04% ± 2.48%	0.99% ± 1.42%	3.30% ± 2.14%	1.92% ± 1.01%
3 False	28.39% ± 18.05%	17.83% ± 11.87%	0.10% ± 0.22%	0.19% ± 0.22%
3 True	0.56% ± 0.72%	0.27% ± 0.29%	0.30% ± 0.45%	0.36% ± 0.24%
4 False	31.09% ± 13.57%	18.83% ± 9.32%	0.00% ± 0.00%	0.08% ± 0.10%
4 True	0.13% ± 0.10%	0.07% ± 0.02%	0.30% ± 0.45%	0.34% ± 0.30%

Config.	WN V2		WN V4	
	Test H@10	Test MRR	Test H@10	Test MRR
2 False	2.59% ± 3.36%	1.30% ± 1.55%	0.05% ± 0.04%	0.06% ± 0.01%
2 True	0.18% ± 0.35%	0.21% ± 0.19%	0.17% ± 0.02%	0.14% ± 0.03%
3 False	5.78% ± 8.54%	3.52% ± 5.26%	0.13% ± 0.07%	0.08% ± 0.03%
3 True	0.52% ± 0.27%	0.40% ± 0.16%	0.06% ± 0.09%	0.08% ± 0.07%
4 False	7.96% ± 5.33%	4.61% ± 3.10%	0.10% ± 0.03%	0.09% ± 0.03%
4 True	0.52% ± 0.26%	0.41% ± 0.14%	0.08% ± 0.14%	0.08% ± 0.08%

Config.	NL-25		FB V3	
	Test H@10	Test MRR	Test H@10	Test MRR
2 False	1.13% ± 0.61%	0.49% ± 0.17%	0.14% ± 0.16%	0.10% ± 0.07%
2 True	1.38% ± 0.52%	0.94% ± 0.17%	0.35% ± 0.07%	0.34% ± 0.09%
3 False	2.78% ± 0.28%	1.15% ± 0.18%	0.16% ± 0.18%	0.11% ± 0.05%
3 True	1.40% ± 1.14%	0.79% ± 0.29%	0.09% ± 0.15%	0.13% ± 0.10%
4 False	1.83% ± 0.85%	0.87% ± 0.37%	0.17% ± 0.13%	0.15% ± 0.12%
4 True	0.58% ± 0.36%	0.57% ± 0.18%	0.03% ± 0.05%	0.08% ± 0.02%

Config.	NELL V4	
	Test H@10	Test MRR
2 False	0.29% ± 0.21%	0.19% ± 0.09%
2 True	0.03% ± 0.04%	0.40% ± 0.05%
3 False	0.37% ± 0.11%	0.21% ± 0.05%
3 True	0.15% ± 0.34%	0.11% ± 0.08%
4 False	0.15% ± 0.24%	0.12% ± 0.14%
4 True	0.34% ± 0.30%	0.30% ± 0.19%

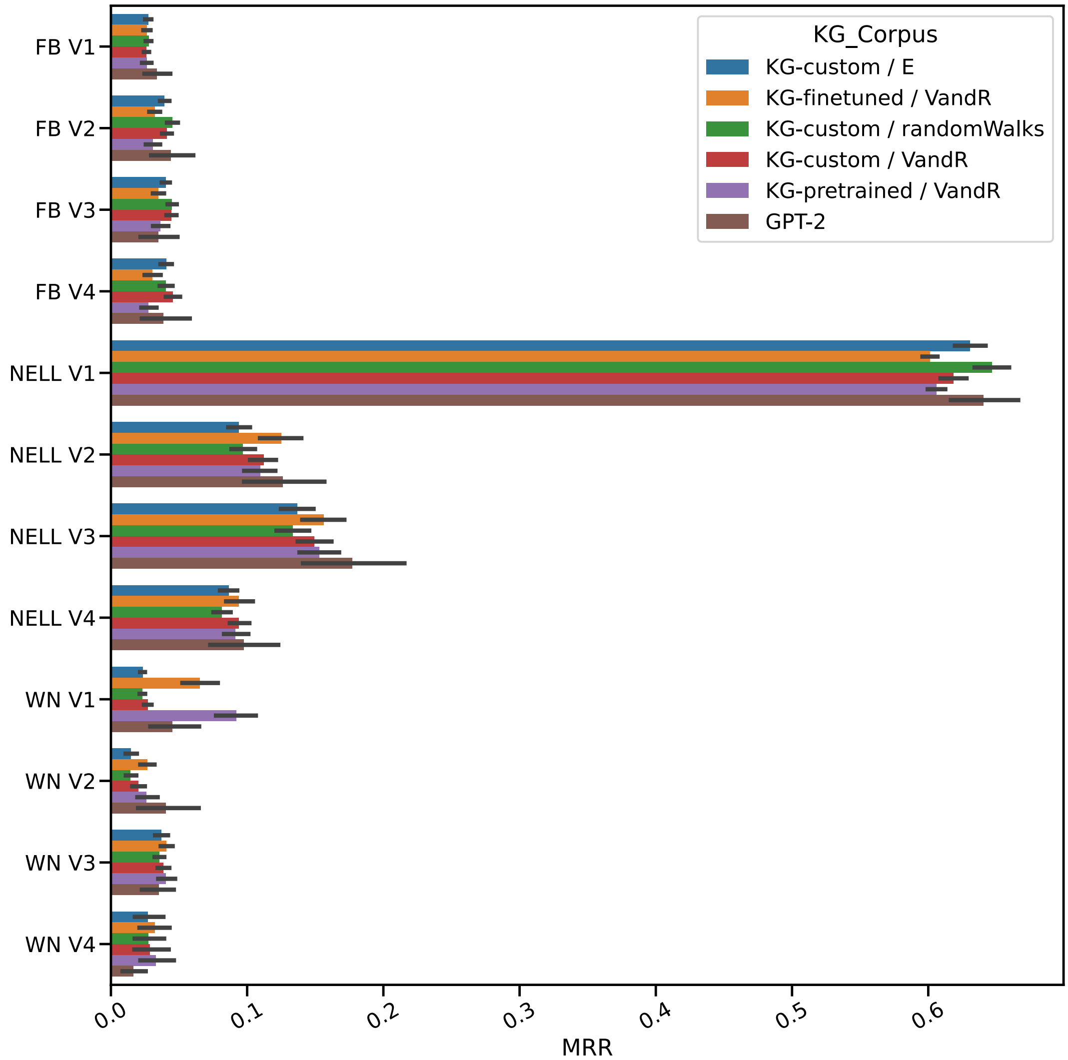


Figure 7: **Custom vocabularies have achieved better results on the inference graphs than the GPT-2 vocabulary.** This figure illustrates the best reciprocal rank (MRR) performance across a diverse range of exploratory experiments. The bars represent the maximum MRRs across all used vocabularies.