Bachelor's Thesis

Improved Simple Question Answering over Wikidata

David Otte

Examiner: Prof. Dr. Hannah Bast
Adviser: Prof. Dr. Hannah Bast

University of Freiburg
Faculty of Engineering
Department of Computer Science
Chair for Algorithms and Data Structures

June 01st, 2023

Writing Period
01. 03. 2023 – 01. 06. 2023

Examiner
Prof. Dr. Hannah Bast

Second Examiner
-

Adviser
Prof. Dr. Hannah Bast

Declaration

Abstract

In Knowledge Graph Question Answering (KGQA), the goal is to answer natural language questions over knowledge graphs. Many previous works focused on question answering over Freebase, which support was cancelled in 2015. Instead, we further investigate question answering over the full Wikidata Knowledge Graph and propose a pipeline for question answering over Wikidata. We focus on simple questions, which are questions that can be answered with a SPARQL query that only uses a single triple. Still, our approach is principally not limited to simple questions and could be extended for complex questions. Given a natural language question, our pipeline generates a set of query candidates, ranks them according to collected features and takes the results of the highest ranked query as the answer to the question. One main focus of our work is the relation matching, where we propose a relation scorer based on the Sentence-BERT architecture for fast and accurate matching. We also discuss the quality of existing question answering datasets for Wikidata and experiment with generating additional questions from Wikipedia articles for fine-tuning. We evaluate our pipeline on three different benchmarks, achieving good results on each. On the SimpleQuestions-Wikidata benchmark, our pipeline achieves an accuracy of 0.816.

Zusammenfassung

Bei Knowledge Graph Question Answering (KGQA) geht es darum, Fragen in natürlicher Sprache über Knowledge Graphen zu beantworten. Frühere Forschung konzentriert sich überwiegend auf die Beantwortung von Fragen über Freebase, dessen Unterstützung im Jahr 2015 eingestellt wurde. Stattdessen untersuchen wir die Beantwortung von Fragen über dem gesamten Wikidata Knowledge Graph und stellen eine Pipeline für diese Aufgabe vor. Wir konzentrieren uns auf einfache Fragen, das sind Fragen, die mit einer SPARQL Query beantwortet werden können, die nur ein einziges Tripel verwendet. Dennoch ist unser Ansatz grundsätzlich nicht auf einfache Fragen beschränkt und kann für komplexe Fragen erweitert werden. Ausgehend von einer Frage in natürlicher Sprache, generiert unsere Pipeline eine Reihe von Queries. Diese werden nach gesammelten Merkmalen sortiert und die Ergebnisse der höchstrangigen Query werden als Antwort auf die Frage genommen. Ein Schwerpunkt unserer Arbeit ist das Relation Matching, bei dem wir einen auf der Sentence-BERT Architektur basierenden Relation Scorer für schnelles und genaues Matching vorschlagen. Wir diskutieren auch die Qualität bestehender Datensätze für die Beantwortung von Fragen über Wikidata und experimentieren mit der Generierung zusätzlicher Fragen aus Wikipedia-Artikeln für das Fine-tuning. Wir evaluieren unsere Pipeline auf drei verschiedenen Benchmarks und erzielen auf jeder dieser Benchmarks gute Ergebnisse. Auf der SimpleQuestions-Wikidata Benchmark erreicht unsere Pipeline eine Genauigkeit von 0,816.

Contents

1. Introduction

Knowledge graphs are widely used to store vast amounts of interconnected data. They mainly organize its data by using the Resource Description Framework (RDF), where statements consist of a subject, a predicate and an object. These are usually referred to as triples or facts. A popular general purpose Knowledge Graph is Wikidata[1], which belongs to the Wikimedia Foundation[2]. Wikidata is the successor of Freebase[3], another general purpose Knowledge Graph, which is not supported anymore since 2015.

Wikidata currently has over 100 million entities, which can be any kind of human knowledge elements. In many practical cases, this can be persons, locations, books and similar things. Information is stored by using properties (the Wikidata term for relations/predicates) to set entities in relation with each other or with other types of data. Since both entities and properties with the same name exist, for example persons with the same name, Wikidata uses unique identifiers. For instance, the Wikidata entity of Barack Obama is `<http://www.wikidata.org/entity/Q76>`. We omit the prefixes of entities and properties and only refer to them with their identifiers, `Q76` is the case of Barack Obama. An example triple statement in Wikidata would be ⟨ `Q76, P106, Q82955` ⟩, where `P106` is the property "occupation" and `Q82955` is the entity "politician".

To acquire specific information from knowledge graphs, often the query language SPARQL (SPARQL Protocol And RDF Query Language) is used. Basic SPARQL queries also use the same triple statement structure. Elements can be replaced with variables, and executing the query gives us the set of possible elements for the selected variables. A SPARQL query for Wikidata that asks for the occupation of Barack Obama would be the following:

```
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
PREFIX wd: <http://www.wikidata.org/entity/>
SELECT ?o WHERE {
  wd:Q76 wdt:P106 ?o .
}
```

[1]https://www.wikidata.org/
[2]https://wikimediafoundation.org/
[3]https://developers.google.com/freebase/

Note that this query, as well as all following queries, makes use of prefixes to abbreviate the full names. Formulating the SPARQL queries for finding specific information can be very challenging for non-experts. One needs to have knowledge about the demanded entity and relation identifiers, the internal structure of the knowledge graph and the structure of SPARQL queries. Knowledge Graph Question Answering (KGQA) is a research field that aims to solve these problems by directly translating a natural language question to the SPARQL query that answers the question. Many proposed systems, as well as question answering datasets, were built for Freebase. Our goal is to further explore question answering over Wikidata and to use deep learning techniques for finding the optimal answer query. For simplicity, we focus on simple questions, which are questions that can be answered with a query that uses only a single triple pattern. However, our approach can also be extended for questions that require queries that use more than one triple pattern. These questions are called complex questions.

1.1. Problem definition

Given a natural language question q, we want to find the SPARQL query s that answers the question q using a single triple pattern. Thus, there are two possible query templates, which we will name analogously to Goette [1]. The first one, we call it Entity-Relation-Target-Pattern (ERT-Pattern), asks for the object of a knowledge graph triple. The following query is an example, it searches for the place of birth (`P19`) of Albert Einstein (`Q937`):

```
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
PREFIX wd: <http://www.wikidata.org/entity/>
SELECT ?o WHERE {
  wd:Q937 wdt:P19 ?o .
}
```

The second one, the Target-Relation-Entity-Pattern (TRE-Pattern) asks for the subject of the triple. The following query is again an example, it asks for entities whose place of birth (`P19`) is Ulm(`Q3012`):

```
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
PREFIX wd: <http://www.wikidata.org/entity/>
SELECT ?s WHERE {
  ?s wdt:P19 wd:Q3012 .
}
```

1.2. Our approach

Our approach builds up on the question answering system Aqqu from Bast et al. [2]. Aqqu was proposed in 2015 for question answering over Freebase. It consists of a pipeline that processes the question to collect features, which are later used to learn a ranking of all the queries that might answer the question. It also supports questions, which can only be answered with using more complex queries. Goette [1] proposed a new version of Aqqu for simple question answering over Wikidata. He uses a similar pipeline but focuses on the collection of basic features. Our approach builds up on the original Aqqu design and the structure relies heavily on the implementation of Goette. Therefore, we will also refer to our system as Aqqu Wikidata. However, we rewrote the majority of the code and added new parts.

The general workflow of our pipeline is as follows: Given a natural language question, our system first identifies possible entity mentions using part-of-speech tags and simple heuristics. In a followup step, all possible query candidates based on the detected entities are generated. This is followed by a relation matching step, where the relation of the generated candidate is compared with the question. We collect multiple features for both the entity linking and the relation matching step and finally train a ranker to infer a ranking. We then take the results of the highest ranked query as the answer of our system.

The main focus in our work is the relation matching step. Using natural language, there is often a huge variety of ways a relation can be expressed. To still be able to reliably recognize fitting relations, we use a deep learning architecture called Sentence-BERT [3]. We evaluate our system on three different benchmarks and achieve good results on each.

1.3. Contributions

In this thesis:

1. We present an extendable system for simple question answering over Wikidata.

2. We propose a relation matching method that relies on the Sentence-BERT architecture.

3. We explain how we handle performance issues that arise when working with the full Wikidata Knowledge Graph.

4. We discuss the quality of different datasets for question answering over Wikidata and investigate the possibility of generating additional data for fine-tuning from Wikipedia articles.

2. Related Work

Yu et al. [4] use bidirectional long short-term memories (LSTMs) to match questions to relations. Therefore, they generate representation vectors for a given relation and a given question and then compare these representations. The question representation is generated by using a hierarchical system of two bidirectional LSTMs, with the assumption, that both look at different levels of abstraction. For the relation representation, they use a bidirectional LSTM given both the single words of the relation and a representation of the full relation as input. To measure how good the question and the relation match, they finally compute the cosine similarity between the two representations. They also propose a simple KGQA system that uses an existing entity linker and their relation detection model. They predict the best query by combining the entity and the relation scores. They evaluate their system on both simple questions and complex questions, and report state-of-the-art results. In our approach, we also generate representations of the question and the relation and compare these representations. The main difference is that we use the Sentence-BERT architecture instead of bidirectional LSTMs.

Wu et al. [5] mention that in previous works for answering simple questions, the systems perform good on the test data, but they perform significantly worse on relations not seen during training. They extend the system of Yu et al. by introducing a relation adapter for solving this problem. This adapter uses the relations seen during training to learn a mapping that is reasonable for all relations. Also, they reorganize the SimpleQuestions [6] dataset for investigating how good the predictions are for unseen relations.

Petrochuk et al. [7] find out that the upper bound of the SimpleQuestions dataset is at 83.4%. They propose a system that only uses standard methods for entity and relation matching and set a new state-of-the-art. They use a conditional random field tagger for predicting the most likely entity and a one layer BiLSTM to predict the most likely relation among all possible relations.

Lukovnikov et al. [8] investigate the usage of BERT for simple question answering over knowledge graphs. They also inspect the effect of limited training data. In their work they use a single BERT model to predict entity spans and the relations together with some further simple measures. They compare the results with two methods using recurrent neural networks and report a slight increase in accuracy for both entity span prediction and relation prediction. The main difference to our work is, that their BERT model outputs probabilities for every relation, while our model works with single input pairs. Also, they use a model for entity span prediction and rely on simple heuristics for ranking, while we use a more simple approach for entity matching and train a model for ranking the query candidates.

Gu et al. [9] highlight the importance of generalization to questions that are not similar to the training questions. They focus on i.i.d. generalization, which is that the system should generalize to questions i.i.d. to the training data, on compositional generalization, which is about the new composition of items seen during the training, and zero-shot generalization to answer questions with unseen items. They propose a new large dataset to learn these generalizations. They also build a Seq2Seq model, where encoder and decoder are recurrent neural networks, and they use a BERT model to get the representations for the input tokens of recurrent neural networks (RNNs). The main differences to our approach are the type of the model, the usage of a model for entity linking and their focus on complex questions.

In December 2022, Perplexity AI[1] presented a Twitter search engine Bird SQL. It uses OpenAI[2] Codex to translate a natural language question into a SQL query that answers the question. Although this problem has significant differences to our problem, this was an impressive demonstration of the power of large language models for query generation. Especially, the large number of different Wikidata entities and properties would make such an approach difficult for KGQA over Wikidata. Currently, the Bird SQL service is not available anymore.

[1]https://www.perplexity.ai/
[2]https://openai.com/

3. Background

In our question answering pipeline, we will make use of a Sentence-BERT (SBERT) model for matching questions with relations. This section aims to give a basic understanding of the training procedures we use and to give an intuition for our SBERT model. Therefore, we first give a general overview of neural networks and their training, and then give an introduction to BERT and our specific model type.

3.1. Neural Networks

Loosely inspired by the neurons in the human brain, Artificial Neural Networks (ANN) became a powerful tool in the area of Machine Learning. Given collected data about a problem, ANNs can be trained to learn patterns from this data and to give valuable predictions for new data.

The fundamental building blocks of neural networks are called neurons. The most simple neural network architecture, called feed-forward neural network, consists out of multiple layers of neurons. The first layer is called input layer and the last layer is called output layer. Each neuron is a function, that takes the outputs of all neurons of the previous layer as input and calculates a new output. Given the outputs of the previous layer $x_1, ..., x_n$, weights $w_1, ..., w_n$, a bias b and a function g, the output a of a neuron is computed with

$$a = g\left(\sum_{j=1}^{n} w_j \cdot x_j + b\right).$$

g is called activation function. Without using activation functions, the final model would be linear. An often used activation function is called RELU, where $g(x) = \max\{0, x\}$. It simply drops negative outputs.

Initially, the weights and the biases of each neuron of the network often get initialized randomly. During a training process, these weights and biases can be updated such that the output layer generates useful outputs for a specific task. In supervised learning, we have a labeled dataset that we can use for training the model. For evaluating the performance of the model and for updating the weights accordingly, loss functions are used. These are functions that usually take the model outputs and the expected outputs as input and compute a score that indicates how well the predictions of the model are. For a binary classification task, the labels in the training data are usually 0 or 1 and often the binary cross entropy loss function is used:

$$L_{BCE} = -\frac{1}{N}\sum_{i=1}^{N} y_i \log(\hat{y_i}) + ((1 - y_i)\log(1 - \hat{y_i})).$$

N is the total number of training instances, y_i is the label of the i-th instance and $\hat{y_i}$ the corresponding prediction of the neural network. With using loss functions, we turn the learning process into a numerical optimization problem. The goal is to minimize the loss function by updating the parameters such that the outputs are as close as possible to the correct labels. For minimizing the loss function, optimization techniques like Stochastic Gradient Descent, and newer variations like Adam [10], are used. The basic idea is to compute the gradient of the loss function at the current point, and to then go into the direction of the negative gradient of this point. Therefore, it is necessary to compute the derivative of the loss function with respect to all weights and biases. This can be done efficiently using backpropagation, which makes use of the chain-rule of calculus.

Often, for efficiency reasons, multiple instances are propagated through the network together. This grouping of instances is called batching. During training, the training data is often propagated multiple times through the network. The process of training the model on the training data once is called one epoch. The chosen batch size, number of epochs, as well as other decisions for the model architecture, such as the number of layers, can have a big impact on the performance of the final model. These parameters are called hyperparameters. The process of finding the best hyperparameters is called hyperparameter-optimization.

A problem that often occurs when training a model is called overfitting. This is when the model fits the training data too closely, meaning that it performs great on the training data but becomes too specialized for performing good on unseen data. Therefore, it is important to choose the correct hyperparameters and to evaluate trained models thoroughly.

Beside the feed-forward neural network, there are multiple other types of neural networks. Convolutional neural networks are mainly used for working with images, recurrent neural networks for handling sequential data and transformers to work on natural language tasks. They all build up on the basics previously explained, but can consist of more complex architectures.

3.2. BERT

The type of neural networks that we will work with are transformers, which were introduced by Vaswani et al. [11]. They rely heavily on the self-attention mechanism, which allows the model to focus more heavily on different parts of the input text. Compared to recurrent neural networks, which were often used for similar tasks, transformers are much faster to train on a GPU because the computations for the whole sequence can be parallelized. A Transformer is composed of an encoder and a decoder. For our use case, we will only make use of the encoder part of the transformer, since we do not want to generate text sequences. An encoder mainly consists of the multi-head attention mechanism and a feed-forward neural network. The exact technical details are not relevant for understanding our approach.

Bidirectional Encoder Representations from Transformers (BERT) from Devlin et al. [12] is a language representation model that consists out of encoders. By stacking multiple encoders, BERT achieved new state-of-the-art results on eleven natural language processing tasks.

One generally distinguishes between pre-training and fine-tuning. During pre-training, the goal is to learn a general understanding of natural language with huge amounts of unlabeled data. In the case of BERT, it was pre-trained on predicting masked tokens in a sentence as well as on next sentence prediction. During fine-tuning, a pre-trained model is taken as starting point. A new output layer is added to the model architecture that fits the task specifics, and the model is trained on new labeled data for a specific problem. The main advantage here is, that due to the pre-training, the model already has a basic understanding of natural language and with that, training is much faster.

For the input, BERT uses WordPiece embeddings. Given one or two sentences, they are splitted into smaller parts given a vocabulary of 30,000 tokens. Also, special tokens such as [CLS] at the beginning of the sequence or [SEP] to separate multiple sentences are inserted. Because of the huge amount of possible words, the idea is to split more complex words into word pieces such that there is a token in the vocabulary that represents this word piece. These tokens then have corresponding vector representations, called embeddings, which are used as numerical input for the network.

3.3. Sentence-BERT

BERT has achieved new state-of-the-art results on sentence-pair regression tasks. Therefore, a cross-encoder architecture was used, where all possible sentence-pairs are fed to the BERT model together and a similarity score gets computed. If the set of sentences gets larger, a huge computational overhead is the consequence. Let

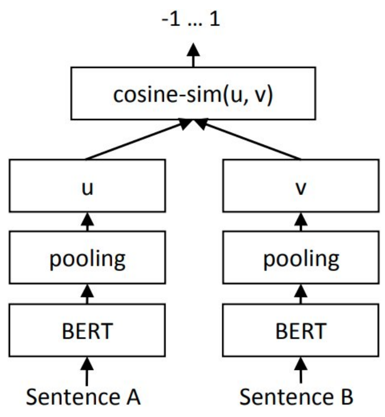

Figure 1.: SBERT model architecture during inference. Figure from Reimers et al. [3].

n be the number of sentences in a given set, then the number of possible sentence pairs, that needs to be fed to the network, computes as $\#pairs = n \cdot (n-1)/2$. For example, when $n = 10,000$, the number of pairs is almost 50 million. Thus, simple inference tasks can take way too long, even on a GPU. [3]

To solve this problem, Reimers et al. [3] propose Sentence-BERT (SBERT), a modification of the pretrained BERT network using a siamese or triplet network architecture. The general idea is, that both sentences are fed separately into the network such that sentence-embeddings are generated, and then the two embeddings are compared using cosine-similarity. Thus, each sentence of a given set of sentences only needs to be processed once. The similarities of all sentence-pairs can then be determined by comparing the embeddings, which is much more efficient. Figure 1 shows the architecture of an SBERT model during inference. Given sentences A and B, both are separately processed with BERT. The two BERT models shown in Figure 1 share its parameters. For every token of the inputted text, BERT outputs an updated embedding. We obtain the final representations of the two sentences by doing a pooling operation over all output embeddings. For SBERT, usually mean pooling is used, where we simply take the mean of all output vectors. The final similarity score of A and B is then obtained by comparing the two representations using the cosine similarity. Looking at the previous example with $n = 10,000$, the inference time was reduced from 65 hours to 5 seconds, without a loss in accuracy [3].

4. Pipeline

In this section, we will explain our pipeline for answering simple questions over Wikidata. The pipeline is based on the pipeline of Bast et al. [2] and its Wikidata version of Goette [1]. Similar to Goette, we also build our pipeline on top of the natural language processing library spaCy from Honnibal et al. [13]. We begin with an entity linking step (Section 4.1), followed by a step for generating the candidates for the identified entities (4.2). Next, a relation matching step that uses a deep relation scorer is applied for matching the relations of the candidates with the question (4.3). In these steps, we extract useful features for every candidate. We give an overview of all collected features (4.4). Finally, a ranker model is trained on the collected values and all query candidates are ranked to find the most promising answer (4.5).

4.1. Entity linking

During entity linking, our main goal is to identify the relevant entity in a given natural language question. We will use a simple, rule-based approach to identify a set of possible entities E. For reducing the search space, we remove unlikely entities and get the pruned entity set E'.

4.1.1. Entity Index

We build an entity index using RocksDB[1], a fast key-value store. When acquiring all necessary information from Wikidata, we use the same SPARQL query as Goette [1] for getting the aliases (see Appendix A.1). For each entity, we get its labels, aliases and the number of Wikipedia sitelinks in this step. We will use the number of Wikipedia sitelinks as a popularity score for the entities, since this is a useful indicator for how well known a specific entity is. We build a mapping from each word sequence that is a label or alias of an entity to all entities it may describe. We also build a mapping from the entity IDs to all information we have about this entity. We will use this index in the following steps for quickly identifying the entities and for fast access to the corresponding information.

[1]https://rocksdb.org/

4.1.2. Identifying possible entities

Given a natural language question, we first use a basic spaCy pipeline to tokenize the question. Tokenization refers to the process of splitting text into smaller chunks called tokens. These tokens are often words, but there are exceptions, for example contractions or punctuation symbols. If the last token is a punctuation symbol, we remove it, since this gives us no further information. Beside the tokenization, spaCy also assigns part-of-speech tags and collects further useful information that we will use in later steps of the pipeline.

To find all possible entities, we look at all subsequences of tokens that might be an alias of an entity. We assume that neighbored tokens, that are both tagged as proper nouns, belong together. To avoid looking on unnecessary subsequences, we merge these neighbored tokens. Now we look at all subsequences of tokens that have at least length two or have length one and are tagged as noun or as proper noun. We use the entity index to find all entities that have this subsequence as label or as alias. With this procedure, we generate a set E, containing all entities identified by the system.

4.1.3. Entity pruning

In the subsequent steps of the pipeline, we will have many possible query candidates for almost every identified entity. To reduce this number and with it the runtime, we implement a simple entity pruning procedure.

First, we want to limit the number of entities that are matched to a single subsequence. Given a subsequence s of the question, let E_s denote the entities of E that were matched by s. We only keep the entities of E_s that have the highest popularity scores. This is necessary because for some subsequences, that for example only contain an often used surname, the number of matched entities can be very high. For example, for s = "Jackson", we have 4,793 resulting entities. We rank the entities by their popularity score and only keep the ten entities with the highest popularity scores. This is a good threshold according to our experience. The big majority of questions is not affected and when the gold entity is not in the top ten, it is unlikely that its candidates would be ranked high anyway.

We sort the entities left primarily by the number of tokens of the longest subsequence they were matched with, and secondarily by their popularity score. We then only keep the first 50 entities in order to have an upper limit. Again, 50 is according to our experience a good threshold. The majority of the questions is not affected because the number of entities after the first pruning step is often already smaller than 50. We call the set of entities that are left after the pruning steps E'.

4.2. Candidate generation

In this section, we use the identified entities from the previous step to generate all query candidates that might answer the question. Since we are focussing on simple questions, we have to consider all queries using the ERT- or TRE-pattern that contain an entity of E' either as subject or as object.

This step of the pipeline can be very time-consuming because of the huge amount of items in Wikidata. We use a single SPARQL query, see Appendix A.2, to generate all candidates as fast as possible. Therefore, we do an inner GROUP BY on the entity and the property to reduce the time for the subsequent operations. In the original Aqqu implementation, we would also acquire the size of the result set for each query candidate. We will not do that because this significantly increases the runtime and gave no improvements regarding the results in our tests.

Assuming $E' = \{\texttt{Q937}\}$, which is the Wikidata ID of Albert Einstein, 4,190 query candidates are generated in this step. 2,922 of them use the ERT-pattern, 1,268 the TRE-pattern.

4.3. Relation matching

In the next step, we want to derive useful measures that state how well the relation of a query candidate matches the question. First, we collect some basic metrics using the tokens of the question and the relation label and aliases. Then we will present our deep relation scorer as a more powerful approach for relation matching.

4.3.1. Relation Index

Similar to the entity index, we also build a relation index using RocksDB. For acquiring the aliases for all relations, we again use the same query as Goette [1], see Appendix A.3. We precompute the lemmatized forms of all relation labels and aliases using spaCy. The lemmatization of a word is its reduction to its base form, which is achieved by reducing inflectional forms. For instance, the lemmatization of the words “am”, “is”, “are” is “be”. We build a mapping that maps each relation ID to its corresponding label, lemmatized label, number of occurrences in the Knowledge Graph and answer type strings. The answer type strings will be explained in section 4.3.5. We also build a mapping from the relation IDs to the corresponding set of lemmatized aliases.

4.3.2. Basic metrics

We define two literal scores similar to Goette [1]. In the following, we will make use of the lemmatizations of the tokens provided by spaCy. Let Q be the set of lemmatized words of the question and R_i the lemmatized words of the i-th label/alias of the Wikidata relation. Let n be the size of the set of the relation label and its aliases. Then, we define the literal score as follows:

$$\textit{literal score} = \max_{i \in \{1,\dots,n\}} |Q \cap R_i|.$$

Since often simple, meaningless words like "the" or "of" are in both sets, we define a set $QC \subset Q$, which contains the lemmatized words of all content tokens of the question. We define content tokens as the tokens, that have one of the following POS-tags: $\{CD, JJ, JJS, NN, NNS, NNP, RB, VB, VBD, VBN, VBP, VBZ\}$, and whose lemmas are not "be", "do", "go" or "have". Then we define the second measure similar to the first one, but with QC instead of Q:

$$\textit{literal content score} = \max_{i \in \{1,\dots,n\}} |QC \cap R_i|.$$

To also have a basic measure that is not as dependent on the chosen words as the literal score, we also use the similarity score from the original Aqqu [2]. The goal is to evaluate the similarity of the lemmatized words of the relation label and the lemmatized content words of the question. For comparing the similarity, we use 300-dimensional word vectors from Word2Vec, learned from parts of the Google News dataset [14]. Word vectors are representations of words that capture their semantic and syntactic meaning. We will use them for capturing the similarity between words. For a given word a, let $w(a)$ be its word vector. First, we get QC' from QC by removing all words that are not in the vocabulary of our word vectors. We define R as the lemmatized words of the relation label, which are also part of the word vectors vocabulary. Then we can compute the similarities using the cosine similarity as follows:

$$\textit{similarity score} = \max \left\{ \frac{w(q) \cdot w(r)}{\|w(q)\| \|w(r)\|} \;\middle|\; q \in QC', r \in R \right\}.$$

4.3.3. Deep relation scorer using SBERT

The basic metrics can be helpful for many questions, but they depend strongly on the aliases provided by Wikidata. Also, they do not regard the other words in the question, which may not directly express the relation but can be a good hint for what is asked for. For instance, questions like "Who is Stephen Curry?" do not contain words that would directly indicate for which relation it asks for. Still, an often expected answer for such a question would be the occupation (`P106`) of the entity. In the case of Stephen Curry (`Q352159`), this would be "basketball player" (`Q3665646`). The goal is to create a model, that gets the question as input and learns to predict which relations are fitting.

For training the deep relation scorer, we will make use of training data D, which consists of question-triple pairs. The first element of the triple is the entity that should be identified in the question, the second element is the relation the question asks for and the third element is a random element of the results of the correct query. Wikidata has over 10,000 different properties and there is no training data D available, which covers all properties. For still being able to generalize to unseen relations, we decide to turn the problem for finding the best matching relation into a binary classification problem. We use two sentences as input, one for the question and one for the relation. Note that sentence in this context rather refers to some kind of text and not necessarily to a grammatical correct sentence. As output, we expect a score between 0 and 1 that measures how well the input question matches the input relation.

For learning a model with these specifications, we use the SBERT architecture introduced in section 3. Our use case is a bit different compared to typical situations where an SBERT model is used. Instead of comparing every sentence pair of a set of sentences, we have exactly one question-relation-pair for every query candidate. When using a BERT cross-encoder, we need to process a sentence pair for every candidate. For some questions, despite the usage of only the top 50 entities, we have over 3,000 query candidates. To process all sentence pairs would then take longer than 3 seconds on an NVIDIA GeForce RTX 2080. This is of course not optimal for a question answering system that should be as interactive as possible. Therefore, SBERT is a nearby choice. Since it allows us to process the question and the relation sentences independently, we can reduce the number of inference steps significantly. For a question we can have at most 50 entities and thus, at most 50 different question sentences. There are over 10,000 possible relations, each with a possibly different relation sentence. We can easily precompute them after fine-tuning the model, since the embeddings will not change anymore. Therefore, we can reduce the number of inference computations from possible multiple thousands to at most 50 for each question.

We use the pre-trained `bert-base-uncased` from Huggingface[2] for our SBERT model. It makes no difference between uppercase and lowercase letters, consists of 12 encoders and has 110 million parameters in total. We linearly map the cosine similarities we compute to the range between 0 and 1.

[2]https://huggingface.co/

Data type	Number of properties
External identifier	7,951
Item	1,586
Quantity	642
String	326
URL	97
Commons media file	77
Point in time	63
Monolingual text	60
Mathematical expression	36
Property	21
Lexeme	16
Sense	16
Geographic coordinates	11
Form	7
Tabular Data	6
Musical Notation	6
Geographic shape	3

Table 1.: Data types of Wikidata properties and how many properties of each data type exist.

4.3.4. Answer type matching

In the triples of the Knowledge Graph, relations often occur together with subjects and objects of specific types. For example, when looking at the property `P19`: "place of birth", usually the subject is a human and the object is a place. The goal of answer type matching is to know for which type a question is asking for and which relations support this type. If the question would be "Where was Albert Einstein born?", we would directly know that the question asks for a place - and can prefer query candidates with relations that often appear with entities that refer to a place. In the following part, we will explain how we integrate this idea of answer type matching into our relation scorer.

Wikidata properties are assigned to a specific data type, which all objects they occur with must have. Table 1 shows all 17 data types that currently exist. We can see that the majority of Wikidata properties has the data type *External identifier*, which means they connect an entity with some kind of identifier. Of course, some questions might refer to identifiers, but most questions will refer to the next most prominent data types. *Item* refers to all properties that can connect two entities. Properties like `P19`: “place of birth” or `P106`: “occupation” are examples for this data type. Also, the data types *Quantity* and *String* are frequent, the objects are then quantities and strings.

We precompute an answer type string for each relation for both the ERT- and TRE-pattern in the following manner: Given property p, we get its data type with the following SPARQL query:

```
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX wikibase: <http://wikiba.se/ontology#>
SELECT ?type WHERE {
  wd:p wikibase:propertyType ?type
}
```

If p has a data type different to *Item*, we simply take the data type name from Table 1 as answer type string for the ERT-pattern. We do not need an answer type string for the TRE-pattern because we only identify Wikidata entities in the question and not dates or other values. For data type *Item*, we do a more fine-grained answer type matching because this will influence most of the relations asked for in questions. For the ERT-pattern, we will take the label of the most prominent super class of all possible objects for p as answer type string. For the TRE-pattern we do the same but for all subjects instead of objects. For the ERT-pattern, we use the following query:

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
SELECT ?label (COUNT(?i) as ?count) WHERE {
  ?s wdt:p ?o .
  ?o wdt:P31 ?i .
  ?i @en@rdfs:label ?label .
}
GROUP BY ?i ?label
ORDER BY DESC(?count)
LIMIT 1
```

We will add the answer type string of a candidate to the relation sentence that we use as input for the relation scorer. The general idea is that the model can then learn a correlation between the words in the question and the answer type string.

4.3.5. Inputs

As input for the relation scorer, we will use a question sentence and a relation sentence. Both sentences should support a powerful relation matching as good as possible.

When we would take the question as it is as question sentence, the knowledge of where the entity in the question is gets lost. Additionally, there would be the risk of overfitting the entities. Therefore, we try two different strategies: masking and marking the entity in the question. To completely prevent overfitting to specific entities, a possibility is to mask the entity in the question. We do this by replacing the entity mention with "<entity>". This gives us the possibility to focus on the words in the question that can express the relation asked for. However, using this masking, all information about the entity gets lost. For example, it would be hard to predict for what relation the question "How big is the <entity>?" is asking for. When we know that the masked entity in this question is `Q513`: "Mount Everest", we could conclude that the relation asked for is `P2044`: "elevation above sea level". As an alternative variant, we propose marking the position of the entity instead of completely masking the entity. Therefore, we just wrap the entity mention between the less than and the greater than symbol. The marked question sentence of the previous example would be "How big is the < Mount Everest >?".

For the relation sentence, we make use of the Wikidata aliases and the precomputed answer type strings. For a relation r, let $S_r = \{r_1, ..., r_n\}$ be the set that contains its label and all its aliases. We get a pruned set of aliases S_r' by dropping all aliases from S_r that only contain uppercase letters because these are likely to just be abbreviations that are not helpful for the model. If then no aliases are left, we set $S_r' = S_r$. Let $a_{r,t}$ denote the answer type string of r and the candidate template t. Then we get the relation sentence by concatenating the answer type string and all aliases with semicolons: '$a_{r,t}$; r_1; ...; r_i'.

Figure 2 gives an overview of the processing of a question-relation pair with the relation scorer.

4.3.6. WikiQuestions

For fine-tuning our model, we will experiment with using additional questions generated from Wikipedia. The existing datasets for simple questions over Wikidata are either small or with low variance regarding the relations. This makes the process of achieving a good generalization over many simple questions hard. To create more diverse questions with corresponding answers, we use WikiQuestions from Prange [15] as starting point. The original WikiQuestions dataset consists of 4,390,597 questions that also include complex questions. For each question, an answer entity or a short string that describes a date is provided. The dataset was created for Freebase, however WikiQuestions also supports the creation of Wikidata questions. First, we will give

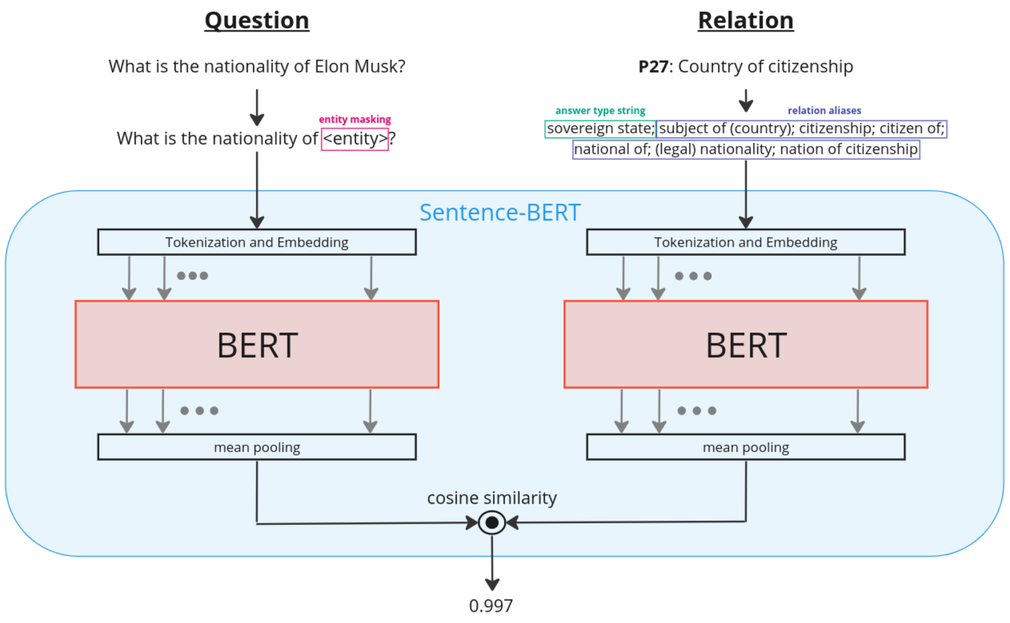
Question
What is the nationality of Elon Musk?
entity masking
What is the nationality of <entity>?
Relation
P27: Country of citizenship
answer type string
relation aliases
sovereign state; subject of (country); citizenship; citizen of; national of; (legal) nationality; nation of citizenship
Sentence-BERT
Tokenization and Embedding
BERT
mean pooling
Tokenization and Embedding
BERT
mean pooling
cosine similarity
0.997

Figure 2.: Example forward pass of the relation scorer given the question "What is the nationality of Elon Musk?" and the relation `P27`: "Country of citizenship".

an overview of how the WikiQuestions dataset was created and then, how we filter the generated questions to get a simple questions dataset.

We start with all Wikipedia sentences and recognized entities in each sentence. These sentences are then parsed using a spaCy parser. Following specific rules, an answer entity or date is selected and a question word (Who, What, Where, Which, When) is determined. Using various transformations, the sentences are turned into questions, that ask for the answer entity or date. Given the generated questions, multiple filtering steps are applied to improve the quality of the final dataset.

For generating simple questions with the corresponding Wikidata triples, we proceed for each generated question as follows. First, we filter out all questions that contain more than 14 words or more than one entity. If the answer is a date, we transform it to the correct Wikidata date format using the parser of the python library dateutil[3]. Then, given the entity of the question and the answer entity or date, we can send a SPARQL query to find all relations connecting these two items. If the answer is also an entity, we send two queries to look both for the ERT- and the TRE-pattern. If there are no results, the question is definitely not a simple question and we can drop it. If there are results, we randomly select one of the relations that connect the two items as relation for our Wikidata triple. When there are both relations for the

[3]https://pypi.org/project/python-dateutil/

ERT- and TRE-pattern, we always select a relation of the ERT-query. Note that choosing a random relation can introduce noise into the training data. For instance, if the entity in the question is a person that was born and died in the same city, we would randomly choose between `P19`: "place of birth" and `P20`: "place of death" for the same question.

Using this procedure, we build a simple question dataset from WikiQuestions. We start with 1,253,066 Wikidata questions and apply our two additional filtering steps. The resulting dataset contains 54,584 questions and their solution triples. The dataset contains 540 different relations. The generation procedure is not perfect and some questions do not make sense. However, we experiment with fine-tuning the model on these questions in our evaluation additional to the fine-tuning described in the next part.

4.3.7. Fine-tuning the SBERT model

We fine-tune the pre-trained SBERT model for matching question sentences with relation sentences. We propose a fine-tuning procedure using the Multiple Negatives Ranking (MNR) loss function from Henderson et al. [16]. For datasets with a low number of different relations, this procedure is not optimal. Therefore, we also propose an alternative fine-tuning procedure that also makes use of additional relations that do not appear in the training data.

The MNR loss function is known for being very efficient while achieving good results on semantic similarity tasks. Since it only needs positive pairs as inputs, we can directly use each question of our training data D together with the corresponding relation, which is given in the triple. First, the set of pairs needs to be divided into batches without any duplicated questions or relations. Let b denote the batch size we choose for fine-tuning. We transform the questions and the relations of each batch to the corresponding input sentences, as explained previously. We generate the question embeddings $\mathbf{q}_1, ..., \mathbf{q}_b$ and the relation embeddings $\mathbf{r}_1, ..., \mathbf{r}_b$ by processing these sentences through the model. For all $i \in \{1, ..., b\}$, the embeddings $\mathbf{q}_i$ and $\mathbf{r}_i$ belong to a correct question-relation pair.

In the following, we will explain how we compute the MNR loss for a given question embedding $\mathbf{q}_i$. We start with computing the cosine similarities of $\mathbf{q}_i$ and all relation embeddings of the batch. With that, we also have scores for $b - 1$ negative question-relation pairs additional to the score of the single correct question-relation pair. Now, we softmax normalize the computed scores and apply cross entropy loss. Since exactly one of the b question-relation pairs is correct, the parts of the incorrect pairs drop out and the cross entropy loss is similar to the negative log likelihood loss. To increase the difference between the computed cosine similarities, we multiply them by a scaling value s. The following formula expresses the described steps to compute the MNR loss for $\mathbf{q}_i$:

$$L_{MNR}(\mathbf{q}_i, \mathbf{r}_1, ..., \mathbf{r}_b) = -\log\left(\frac{\exp(s \cdot sim(\mathbf{q}_i, \mathbf{r}_i))}{\sum_{j=1}^{b} \exp(s \cdot sim(\mathbf{q}_i, \mathbf{r}_j))}\right),$$

where $sim(\mathbf{q}, \mathbf{r})$ stands for the cosine similarity of $\mathbf{q}$ and $\mathbf{r}$:

$$sim(\mathbf{q}, \mathbf{r}) = \frac{\mathbf{q} \cdot \mathbf{r}}{\|\mathbf{q}\|\|\mathbf{r}\|}.$$

For the scaling value s, we choose the standard value $s = 20$ of the implementation in the sentence-transformers[4] library. Note that separately computing the embeddings of the sentences first makes this procedure very efficient. For a batch size of 16, we need to process 32 sentences individually and get $16^2 = 256$ pairs for training. Therefore, the performance usually increases for an increasing batch size. For speeding up the fine-tuning further, we use automatic mixed precision (AMP). This means that automatically only 16 bit floating point values instead of 32 bit floating point values are used for operations that do not need full precision. This usually has no negative effect on the results. We choose appropriate hyperparameters manually according to results on a set of additional questions, called validation set. We fine-tune the model for four epochs and use a batch size of 16. A larger batch size is not possible on the NVIDIA GeForce RTX 2080 we use.

We provide an alternative fine-tuning method for cases, where the training data has not enough variance. If the data only contains few different relations, it is hard to achieve good results with the MNR loss function, since it only uses the relations in the training data. To address this problem, we will add additional negative sentence-relation-pairs. Before the fine-tuning, we need to process the training data with our pipeline. For each question, we take the query candidate that uses the entity and the relation of the triple in the training data as the correct candidate. We will take each question with the relation of the correct candidate as positive pair (label 1) and each question with 50 of the relations of the incorrect candidates as negative pairs (label 0). We limit this number to 50 (or less if there are less) incorrect relations for continuity reasons. Also, using more than 50 incorrect relations, did not lead to better results on the validation data. We upsample the positive pairs such that the number of positive and negative pairs is balanced. For fine-tuning, we will use the contrastive loss function proposed by Hadsell et al. [17]. It increases the distance between the embeddings of negative sentence pairs and decreases it for positive pairs. The loss for a single question-relation pair and their embeddings $\mathbf{q}_i$, $\mathbf{r}_i$ and the corresponding

[4]https://github.com/UKPLab/sentence-transformers

label y_i can be computed with

$$L_{CL}(\mathbf{q}_i, \mathbf{r}_i, y_i) = y_i \frac{1}{2} \|\mathbf{q}_i - \mathbf{r}_i\|_2 + (1 - y_i) \frac{1}{2} max(0, m - \|\mathbf{q}_i - \mathbf{r}_i\|_2)^2.$$

The parameter m controls the influence of negative pairs. When the distance of the embeddings of a negative pair is greater than m, it does not contribute to the loss function. [17] We choose the standard value of the sentence-transformers implementation and set $m = 0.5$. We again use AMP for faster fine-tuning and choose hyperparameters manually. We fine-tune the model for two epochs and use a batch size of 32. The actual effects of the two different variants will be discussed in the evaluation.

For creating the training set for our ranker, we need to process all questions of our training data with the pipeline. To generate relation scores that are not overfitted, we split all candidates into three separate folds. Then, we fine-tune the relation scorer on each pair of these folds and predict the relation scores for the candidates of the third fold. Finally, we fine-tune the relation scorer that will be used during evaluation on all training questions.

4.4. Candidate features

During the previous steps, we have collected different features for each query candidate. These features should contain useful information for how relevant a candidate is. In the next section, we will train a ranking model on these features in order to rank the candidates to predict the most likely answer.

Table 2 gives an overview of all features we collected. Features 1-4 describe the entity linking and features 5-11 the relation matching. The idea is, that when we have good representative features for both entity matching and relation matching, we can easily learn to rank the candidates for finding the best query.

4.5. Ranking

In this section, we will explain how we train and use a random forest model for ranking all query candidates. To reduce the runtime of the pipeline further, we also provide a simple but effective rule for pruning the set of candidates.

For training the ranker, we process all questions of the training data D with our pipeline and leave out the ranking step. We again define the candidate with the intended entity and relation as correct candidate and all other candidates as incorrect candidates. An important question to ask here is, whether other query candidates that also give the correct answer should also be considered as correct candidates.

ID	Description
1	**Exact entity match:** Binary value that is one if a subsequence of the question exactly matches the label of the entity.
2	**Exact entity token matches:** Number of tokens in the question that appear in the entity label.
3	**Entity popularity score:** Number of Wikipedia sitelinks the entity has.
4	**Exact relation match:** Binary value that is one, if the lemma of a relation alias matches the lemmatized question tokens of a question perfectly.
5	**Literal score**
6	**Content literal score**
7	**Exact token matches:** Sum of exact entity token matches and literal score.
8	**Similarity score**
9	**Relation score**
10	**Proportion matched/total tokens:** Number of question tokens that were matched in any way divided by the number of tokens in the question.
11	**Occurrences relation KG:** Number of times the relation occurs in the knowledge graph.

Table 2.: Overview of all features we extract for the query candidates.

This would make sense for some candidates, but there are often queries yielding the correct result, although the used entity is incorrect or the relation has not the intended meaning.

This gives us a situation different to the standard Information Retrieval ranking situations. Thus, classical learning to rank approaches may not be appropriate for our ranking problem. Bast et al. [2] investigate different simple ranker models, from which we will choose the best performing one - the pairwise random forest ranker. First, we reduce the number of candidates, such that questions with many candidates have not a too big influence on the ranker. We do this by only taking half of the candidates, 200 if the half would be smaller than 200 or all candidates if there are less than 200. We call the set of feature vectors of the incorrect candidates $C' = \{c_1, ..., c_n\}$ and the feature vector of the correct candidate c. As training data for the ranker, we will create pairs of these vectors and corresponding labels that state whether the candidate of the first vector should be ranked higher than the candidate of the second vector. We create the training data for the ranker by taking all positive pairs $(c, c_i, c - c_i)_{i=1}^{n}$ with label 1 and all negative pairs $(c_i, c, c_i - c)_{i=1}^{n}$ with label 0. We train a random forest with 100 decision trees on this data.

When processing questions with our pipeline, we have around 400 candidates on average per question, sometimes even over 2,000. Since the time for the ranking increases significantly with an increasing number of candidates, we try to remove candidates that are very unlikely to give the correct answers. In our case, we do this by a simple rule: We remove all candidates with a content literal score of zero and a relation score that is smaller than 0.5. This effectively removes more than half of the candidates on average without decreasing the accuracy of our system. One could also increase the bound of 0.5, however, this increases the risk of falsely removing a correct candidate.

For creating the final ranking, we first create the input vectors for each pair of candidates as described previously. We then process these vectors through the model and sort the candidates by their number of "won" comparisons. We can now execute the highest ranked query and take its result as answer of our system. We also acquire the labels of the results if they exist. For the example query in section 1.1, that asks for the birthplace of Albert Einstein, we would finally execute the following query:

```
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
PREFIX wd: <http://www.wikidata.org/entity/>
SELECT DISTINCT ?o ?label WHERE {
  wd:Q937 wdt:P19 ?o .
  OPTIONAL {
    ?o @en@rdfs:label ?label .
  }
}
```

Note that we make again use of the QLever-specific syntax `@en@`. For other backends, one can simply use an additional `FILTER` step for only getting the English label.

5. Evaluation

In this section, we will evaluate our system on three different benchmarks. First, we give an overview about which hardware and SPARQL backend we use (5.1). Then, we give a short introduction to the three benchmarks (5.2) and explain our evaluation measures (5.3). We discuss our results on the benchmarks and experiment how the results change when using different configurations of our pipeline (5.4). Finally, we analyze the concrete errors of the pipeline on the three benchmarks (5.5).

5.1. Hardware and SPARQL Backend

The evaluation was done on a server with an AMD EPYC 7502 32-Core Processor, 64 GB RAM and an NVIDIA GeForce RTX 2080 GPU. As SPARQL backend, we use QLever, a very fast SPARQL Engine introduced by Bast et al. [18]. Our instance uses the Wikidata dump from 25.05.2023, has 18,402,458,084 triples in total and is running on an SSD. To reduce the influence of varying load on the QLever backend on the evaluation, we average all runtimes over two runs.

All precomputation steps can be done in 45 minutes. This includes acquiring entities and relations from Wikidata, precomputing lemmatizations and answer type strings, and building the indices. Processing around 20,000 training questions needs roughly 4 hours. All following training and prediction steps until the pipeline is ready need 9 hours when using the contrastive loss function and 5 hours when using the MNR loss function.

5.2. Datasets

Most of the existing KGQA datasets were either created for other knowledge graphs like Freebase, or focus on complex questions. In the following, we describe the datasets we use for evaluating our system.

Dataset	#q train	#r train	#q test	#r test	new relations
SimpleQuestions-Wikidata	19,481	76	5,622	72	0
LC-QuAD 2.0 SQ	6,261	2,845	824	794	593
Own questions	35,009	3438	50	43	6

Table 3.: Overview of the benchmarks for the evaluation. #q is the number of questions, #r is the number of unique relations. New relations refers to the number of relations of the test set, that do not occur in the training set.

5.2.1. Overview

We use three different benchmarks: SimpleQuestions-Wikidata, LC-QuAD 2.0 SQ and a small set of own questions. SimpleQuestions-Wikidata [19] is the only existing larger simple question dataset for Wikidata. LC-QuAD 2.0 [20] is a dataset for question answering over Wikidata that also contains many complex questions. We will use the simple questions of this dataset and call it LC-QuAD 2.0 SQ. Additionally, we provide a small own benchmark containing 50 different questions.

Table 3 gives an overview of the three benchmarks. Beside the sizes of the training and test sets, we also provide the number of different relations they contain. This number is very important for creating a relation scorer that is able to generalize. We can see big differences between the datasets. While the LC-QuAD 2.0 SQ training set has 6,261 questions with 2,845 different relations, the SimpleQuestions-Wikidata training set contains 19,481 questions but only 76 relations. As Wu et al. [5] point out, many question answering systems perform good on the test data but significant worse on questions requiring other relations. For being able to better analyze the generalization capabilities of our pipeline, we also provide the number of relations in the test sets, that do not occur in the training sets. We can see that the SimpleQuestions-Wikidata test set has no unseen relations at all, while the majority of questions in the LC-QuAD 2.0 SQ test set uses unseen relations.

5.2.2. SimpleQuestionsWikidata

A well known dataset for simple questions is the SimpleQuestions dataset [6]. It was designed for question answering over Freebase and contains 108,442 simple questions written by human annotators.

Diefenbach et al. [19] port the SimpleQuestions dataset to Wikidata by mapping subjects and objects to Wikidata items with an automatically generated mapping and by using a handmade mapping for the relations. Although the SimpleQuestions dataset contains 1,837 relations, they only map 404 of them to Wikidata, since they already cover a big amount of the questions. The resulting dataset is called SimpleQuestions-Wikidata and contains 49,202 questions, from which 27,924 are answerable over Wikidata. We will only use these answerable questions for our evaluation.

Due to the mapping of the relations, the answerable questions only cover 76 relations overall, which is much smaller than the 1,837 in the SimpleQuestions dataset. Since 76 relations are only a very small subset of the over 10,000 relations in Wikidata, this can be critical for training the relation scorer. Also, all relations in the test set also appear in the training set, what makes it impossible to test the generalization abilities of the relation scorer.

5.2.3. LC-QuAD 2.0

LC-QuAD 2.0 is the second version of a large scale question answering dataset for complex questions. It was created by Dubey et al. [20] and consists of 30,000 questions with gold queries provided for both Wikidata and DBpedia. The question generation workflow starts with selecting entities with popular Wikipedia sites. A set of SPARQL query templates is defined such that many different types of questions can be covered. Depending on the selected template, a set of predicates is selected and a subgraph of the KG is generated. With combining the template and the subgraph, the final SPARQL query is generated. This query is then transformed into a natural language template called question template. For formulating the natural language question from this question template, they use a large crowdsourcing experiment consisting of three steps. In a first step, the so called turker formulates a question out of the question template. In a second step, for many questions an alternative, paraphrased question is formulated and finally, the formulations are checked to ensure quality.

For our evaluation, we will focus on the simple questions of LC-QuAD 2.0. Therefore, we have to distinguish between questions regarding a single fact and questions regarding a single fact with type. A question using a single fact with type would be "Which public holiday celebrates the birth of Jesus?", the corresponding SPARQL query would be

```
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
PREFIX wd: <http://www.wikidata.org/entity/>
SELECT ?s WHERE {
  ?s wdt:P547 wd:Q51628 .
  ?s wdt:P31 wd:Q1197685 .
}
```

with `P547`: “commemorates”, `Q51628`: “Nativity of Jesus”, `P31`: “instance of” and `Q1197685`: “public holiday”. For generating our training data, we will use the provided training set. We use all single fact questions as well as all single fact with type questions. Although our system is not built for answering questions with an additional type, the questions are often very similar to questions that we consider as simple questions. This doubles the number of questions we can use for training. For the single fact with type questions, we simply remove the triple checking the type of the answer. Concretely, we use all questions of the training set, where the subgraph is either “center”, “simple question left” or “simple question right”. This gives us a training set with 6,261 questions.

Analogously, we will use the provided test set for generating our test data. Here, we will only use the single fact questions, since the single fact with type questions would optimally require a different query template. Therefore, we will use all questions, where subgraph is “center”. To prevent confusion, note that all questions marked with subgraph “simple question left” or “simple question right” are questions asking for a single fact with type. The resulting test set contains 824 questions.

For both the training and the test set, we observe that the majority of the questions ask for identifiers. This increases the number of used relations, but may be untypical for questions the system will answer in practice.

5.2.4. Own questions

Additionally to the two previously described benchmarks, we also provide 50 own questions to evaluate our system. These questions are all answerable and formulated with as much variation as possible. We already used these questions during the development of the system, we are aware that this might lead to overfitting. However, we did no concrete steps to perform better on these questions and think that the results are representative for other simple questions. For training the model, we combine the full datasets SimpleQuestions-Wikidata and LC-QuAD 2.0 SQ.

5.3. Evaluation measures

For evaluating our system, we measure the accuracy and the top-k scores on the benchmarks.

Accuracy: Let G_i be the set of the results of the gold query for question i and C_i the set of results of the highest ranked query candidate. Then we define the accuracy as the percent of questions for which the predicted answers are the same as the gold answers:

$$accuracy = \frac{1}{n} \sum_{i=1}^{n} I(G_i = C_i),$$

with I being the indicator function.

Top-k: The Top-k score is the fraction of questions, where one of the k best ranked query candidates returns the set of answers of the gold query. Top-1 equals the accuracy.

5.4. Results

As standard configuration of our system, we use all features listed in Table 2 and choose the parameters as explained in section 4. For fine-tuning the relation scorer, we will use the MNR loss function and entity masking for creating the entity sentences as standard. Due to the varying quality of the questions in the WikiQuestions dataset, we will not use it in the standard configuration. We evaluate our system on the three benchmarks. For the SimpleQuestions-Wikidata dataset, we will use the contrastive loss function for fine-tuning the relation scorer. As pointed out in section 4.3.7, the MNR loss function would not be optimal with a small amount of different relations in the training data. For the evaluation, we limit the number of considered answers per query to 1,000. Queries using the TRE-pattern can possibly have millions of answers. In these cases, it would be very expensive to then compare the whole answer sets. However, this should only make a minor difference. It only influences a subset of the training questions and also then, comparing the first 1,000 answers of the queries is usually representative for comparing the whole answer sets.

Table 4 shows the accuracies and the top-k results of our pipeline on all three benchmarks. We see that our pipeline is able to answer a big majority of the questions correctly and achieves an accuracy of over 0.81 on all three datasets. Looking at the top-k results, we see that in many cases when the predicted answers are not correct, the next highest ranked candidates provide the correct results. In 86% of the cases, the correct candidate is among the two highest ranked candidates for all benchmarks. The average runtimes for the questions lie between 0.4 and 0.6 seconds, which can be considered as interactive.

Table 5 compares the results of our pipeline on the SimpleQuestions-Wikidata benchmark to the results of other QA systems on the SimpleQuestions-Wikidata and the SimpleQuestions benchmark. While we and Goette use the accuracy defined above, the other QA systems compare the queries for evaluation. This might lead to a slightly lower accuracy for their systems. With an accuracy of 0.816, our system achieves the highest accuracy. It is very important to point out that it is hard to compare the results on SimpleQuestions-Wikdiata to the results on SimpleQuestions. For

Dataset	Accuracy	Top-2	Top-3	Top-5	Top-10	AD
SimpleQuestions-Wikidata	0.816	0.863	0.879	0.889	0.895	0.49
LC-QuAD 2.0 SQ	0.825	0.860	0.865	0.873	0.877	0.57
Own questions	0.820	0.880	0.920	0.960	0.960	0.46

Table 4.: Results of Aqqu Wikidata on our three benchmarks. AD is the average duration of the questions in seconds.

QA System	SimpleQuestions (FB2M)	SimpleQuestions-Wikidata
Yu et al. (2017)	0.787[1]	-
Petrochuk et al. (2018)	0.781	-
Huang et al. (2019)	0.754	-
Lukovnikov et al. (2019)	0.773	-
Oliya et al. (2021)	-	0.682
Goette (2021)	-	0.586
Aqqu Wikidata (2023)	-	0.816

Table 5.: Accuracy of our pipeline on the SimpleQuestions-Wikidata benchmark compared to the accuracies of other QA systems.

SimpleQuestions-Wikdiata, we have only 76 instead of over 1,000 different relations and also have no unseen relations in the test data. We see a huge gain in accuracy of our pipeline compared to the version of Goette. One main factor for this improvement is the relation scorer. As already mentioned in section 2, the upper bound of the SimpleQuestions dataset is 83.4%. Achieving a higher accuracy is not possible due to errors in the dataset and answers that can not be disambiguated. There is no specific upper bound known for the SimpleQuestions-Wikidata dataset, however an upper bound in a similar range can be expected.

We experiment with different configurations of our pipeline to investigate how these changes influence our results. Table 6 shows the configurations together with the achieved accuracies and the average runtimes per question. Compared to the standard configuration, leaving out the relation score leads to significant loss in accuracy on both SimpleQuestions-Wikidata and our own benchmark, while the difference on LC-QuAD 2.0 SQ is only small. The main reason for this is that the relation scorer has problems with abbreviations, which often occur in questions asking for identifiers.

[1]Yu et al. [4] do not provide a publicly available implementation of their system. Huang et al. [21] mention that they were not able to replicate the reported results.

	SimpleQuestions-Wikidata	LC-QuAD 2.0 SQ	Own questions	AD
Full Pipeline	0.816	0.825	0.820	0.50
w/o rel score	0.673	0.808	0.760	0.44
w/o rel occs, w/o sim score	0.811	0.823	0.760	0.40
only rel and popularity score	0.792	0.785	0.740	0.38
entity sentence: marking	0.795	0.826	0.820	0.59
fine-tuning WikiQuestions	0.813	0.823	0.820	0.52
entity pruning: 200/500	0.818	0.819	0.820	1.76
no candidate pruning	0.816	0.825	0.820	2.01

Table 6.: Detailed analysis of our pipeline by evaluating its accuracies on the three benchmarks using different configurations. For the variant using WikiQuestions, we fine-tune the model for two epochs using the MNR loss function before the actual fine-tuning step. For the more lenient entity pruning step, we limit the maximal number of entities per sequence to 200 and the total number of used entities to 500[2]. AD is the average duration of all questions of the three benchmarks in seconds.

This will be discussed more in depth in the next section. We also experiment with only using the relation scorer, together with the popularity score in order to have some way to prefer entities to other entities. Despite dropping all other features, the pipeline is still able to achieve an accuracy of 0.74 or more on all three benchmarks. We can see that both variations of the fine-tuning, marking the entity in the entity sentence and additionally fine-tune on WikiQuestions, do not really give any improvement. Only for the LC-QuAD 2.0 SQ there is a small increase in accuracy when marking the entity. Still, we report the results of the full pipeline as our final results in Table 4 for having a consistent pipeline configuration. For all these variations, the questions can be answered in under 0.6 seconds on average. Both allowing more entities and removing the candidate pruning step lead to no significant improvement but increases the average runtime to 1.76 seconds respectively 2.01 seconds.

Table 7 shows the accuracies on the SimleQuestions-Wikidata and the LC-QuAD 2.0 SQ benchmarks for both fine-tuning approaches. As the results show, our intuition was correct and we chose the best fine-tuning approaches for both benchmarks. We can see that the accuracy on the LC-QuAD 2.0 SQ dataset is still very high when using the contrastive loss function, however looking at Table 6 we see that it is not better than using no relation scorer at all. On the SimpleQuestions-Wikidata benchmark,

[2]Since a single query for candidate generation with 500 entities would be too long, we use multiple queries using at most 50 entities.

	SimpleQuestions-Wikidata	LC-QuAD 2.0 SQ
MNR loss fine-tuning	0.799	0.825
contrastive loss fine-tuning	0.816	0.807

Table 7.: Results on SimpleQuestionsWikidata and LC-QuAD 2.0 SQ for the two different fine-tuning approaches.

we achieve an accuracy of 0.799 when using the MNR loss function. After first tests, we did not expect that this result will be so close to the result of fine-tuning with the contrastive loss function. This shows, that already a small amount of different relations can be sufficient for fine-tuning the relation scorer appropriately with the MNR loss function. Still, we can see that the contrastive loss fine-tuning performs better than the MNR loss fine-tuning when having a small amount of relations.

5.5. Error analysis

We inspect the errors of our pipeline by hand by looking on big enough subsets of our benchmarks. On each of the three benchmarks, there are different main sources of errors. In the following, we will explain them for each benchmark and provide some examples.

SimpleQuestions-Wikidata

On the SimpleQuestions-Wikidata benchmark, for around 10% of the questions the correct entity could not be identified. Without the correct entity, it is not possible for the system to generate the candidate that represents the gold query. The results of the highest ranked query could still be correct, but this is almost never the case and not the intended way to answer questions. There are two main reasons for the correct entity not being identified. The first reason are errors in the dataset. One example would be the question “who wrote \\"w\\" is for wasted” from the benchmark, where backslashes were added, which makes it impossible to match it with “"W" is for Wasted” (`Q13422918`). A different example is the question “Name a model that died due to a car accident”, which is not even a simple question. The expected answer in the dataset are just all people that died during a car accident. However, it is not possible to identify the correct entity `Q9687`: “traffic collision” because it has no alias called “car accident” or “accident”. The other main reason for the correct entity not being identified are mistakes in the POS-tagging of the pipeline. For considering a single token as entity label, it must be either tagged as a noun or a proper noun. This is for example not the case for the question “What’s a gameplay mode in sacred” and the correct entity “Sacred” (`Q1757845`).

Another often occurring source of error is that the best ranked candidates only differ in their used entity. The relation score is then often the same because the same relation is used despite the different entity. The only way for us to then infer a ranking is to use the popularity scores of the entities. When the correct entity has a lower popularity score than the entities of other candidates, the correct candidate will not get ranked at first place. For example, when processing the question "What position does Carlos Gomez play?", we do not know which of the currently 17 Carlos Gomez entities in Wikidata the question refers to. The correct Carlos Gomez according to the dataset is `Q2747238` with a popularity score of 7, while there is also a Carlos Gomez `Q203210` with a popularity score of 16. So for these questions, the error also lies in the benchmark. In very few cases, the question in such a situation is not answered correctly due to mistakes of the ranker. Then, two candidates with the same features except of the popularity score are ranked in the wrong order.

LC-QuAD 2.0 SQ

There are two main reasons for errors on the LC-QuAD 2.0 SQ benchmark. The first one is again the missing identification of the correct entity in the question. Similar to our findings on the SimpleQuestions-Wikidata benchmark, this is either due to incorrect POS-tagging or to errors in the questions of the benchmark. A different, often occurring problem are bad predictions of the relation scorer for questions asking for identifiers. Especially on the LC-QuAD 2.0 SQ dataset, we have a big amount of questions asking for identifiers. These questions often have abbreviations in the question that the relation scorer cannot really work with. Sentence-BERT relies on WordPiece tokens, which can create difficulties when using a seemingly arbitrary sequence of letters. Thus, even when the label of the correct identifier relation appears similar in the question, other identifier relations that should not match get a higher score. An example would be the question "Which is NLBPA ID for Hank Aaron?", where the correct property "NLBPA ID" (`P4405`) has a relation score of 0.797, while the incorrect property "Bibliothèque nationale de France ID" (`P268`) has a relation score of 1.

Own benchmark

On our own benchmark, we have no problems with errors in the dataset. For one question, the candidate generation query failed because it exceeded the available memory of our SPARQL backend. For all other questions of the benchmarks, as well as for the same question in earlier tests, this was no problem. The other questions that were not answered correctly can be divided into two groups. The first group are questions that ask for answer types that do (almost) not occur in the training data, such as numbers, coordinates or dates. Example questions are "How many people live in Bavaria?" and "What is the location of the Golden Gate Bridge?", which both are not answered correctly. The other group of questions consists of questions that are more colloquial and often do not use words that express the relation directly. "How big is the Mount Everest?" would be an example for such a question. Even though the pipeline has problems with these questions, often they are still answered correctly. The deciding factor for these questions is the training data, if we can create more diverse datasets, these questions could be answered more reliably.

6. Conclusion

In this work, we presented an expandable system for simple question answering over Wikidata. For relation matching, we proposed an SBERT model, which effectively learns how to match questions with relations. Due to its efficiency, as well as optimization of the SPARQL queries and simple pruning steps, our pipeline is able to answer questions over the full Wikidata Knowledge Graph in around 0.5 seconds on average. Especially because of the relation scorer, we could significantly increase the accuracy of the pipeline compared to the version of Goette [1]. We discussed the quality of different simple question benchmarks and evaluated our pipeline thoroughly. We achieved good results on all three benchmarks we evaluated our system on.

6.1. Future work

Our pipeline already can answer the majority of simple questions correctly. However, there are different possibilities to further improve the results on simple questions and to further extend the system. In the following, we will give an overview of the most important possible improvements.

6.1.1. Improved entity matching

Our entity linking procedure is very simple and not fully reliable, as the evaluation shows. Also, it gives us no preference among the entities, so we have to rely heavily on the popularity score. A simple approach would be to use a BERT model for predicting the entity spans. This could be done similarly to Lukovnikov et al. [8]. Also, some robustness to spelling errors would be very helpful. However, this can also increase the number of possible entities. With focusing more deeply on the entity matching, it would also make sense to expand the evaluation frontend of Goette [1] for also directly evaluating the results of the entity linking step. Estimated time: 4 weeks.

6.1.2. Working with complex questions

For simplicity, we focus on simple questions in this thesis. However, many real world questions involve multiple entities in the question and, thus, more complex query patterns. First, we would need to adapt the candidate generation step for also generating other queries. As in the original Aqqu, the most straightforward approach would be to increase the set of query patterns we use. This approach results in a big amount of query candidates and limits us to the query patterns we include. Regarding the huge successes of large language models in recent years, it would be very interesting to train them on predicting a query pattern for a given question. As mentioned in section 2, Bird SQL is an example of a tool that already demonstrated the abilities of language models for generating queries. If the resources for large language models are not available, also smaller sequence-to-sequence models could give valuable predictions. We would also need to adapt the relation scorer for handling multiple relations at once. Therefore, one could either input a concatenation of all relations or take the mean of the single relation embeddings. Estimated time: 3 months.

6.1.3. Faster runtime

Although we can answer questions in around 0.5 seconds on average, there is still space left for improvement. Especially, it would be very helpful if we could achieve a consistent runtime of under one second for each question, which is currently not the case. As already mentioned, the biggest bottleneck regarding the runtime is the candidate generation step. One could expect that simply getting the relations of 50 entities together with the subject/object information should be possible in a consistent and short time period. Maybe, future work on QLever can achieve that. Another approach could be to precompute all candidates for each entity. However, this approach would not be applicable to complex questions and thus is not interesting for us, since we want to provide an extendable system.

Since the focus was to reduce the runtime with an efficient relation matching and optimizations regarding the candidate generation, there is still room for improvement in other parts of the pipeline. Beside an improved entity linking step, also a candidate pruning that makes use of a trained model instead of a simple rule could reduce the runtime significantly. Estimated time: 2 weeks.

6.1.4. Generating better training data

As mentioned in the evaluation, there is a lack of datasets for simple question answering over Wikidata. One way to generate more training data would be to take question answering benchmarks for Freebase, and to translate them to Wikidata.

There already exist mappings from Freebase entities to Wikidata entities which we can use. Using a handmade mapping between the relations, as [19] did, only limits the number of relations in the training data. Therefore, we would suggest sending a SPARQL query for each question using the subject and the object and searching for relations that connect them in Wikidata. Then we can randomly take one of these relations for the dataset. Doing this, we would have the same problem that we already mentioned in section 4.3.6. The random choice of the set of relations can lead to an unintended relation and introduce noise in the data.

Another interesting approach would be to update the question generation process of WikiQuestions. Currently, many questions do not make sense or are not correct. Improving the quality of the generated questions, we definitely see potential in generating questions from Wikipedia sentences. Also, one could make again use of deep learning to achieve a more grammatical correct reformulation of the sentences. Estimated time: 2 months.

7. Acknowledgments

First, I would like to thank Prof. Dr. Hannah Bast for the very valuable advice and helpful discussions during the writing process. Her feedback was always constructive and very motivating. Many thanks to Johannes Kalmbach for always being available when there were problems with QLever or I needed help with query optimizations. Thanks also to Natalie Prange for her advice on WikiQuestions. Thanks to Johannes for reading my thesis. Finally, I would like to thank my family and friends for always supporting me during a bit stressful times.

A. Appendix

A.1. Entity index query

We use the following SPARQL query to get all entity aliases for our entity index. Table A1 gives the corresponding labels to the used properties. Note that this query uses `@en@`, which is a QLever-specific [18] shortcut for restricting the results to those in English language. [1]

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
SELECT DISTINCT ?entity ?alias WHERE {
    ?entity (wdt:P734)?/(@en@rdfs:label|@en@skos:altLabel|@en@wdt:P74⌋
    ↪  2|@en@wdt:P1448|@en@wdt:P1449|@en@wdt:P1477|@en@wdt:P1559|@en⌋
    ↪  @wdt:P1705|@en@wdt:P1813| wdt:P297|wdt:P298|wdt:P1160) ?alias
    ↪  .
        MINUS {
            # Ignore items internal to wikidata (around 7M)
            ?entity wdt:P31/wdt:P279* wd:Q17442446 .
        }
    }
ORDER BY ASC(?entity) ASC(?alias)
```

Property ID	Label
P734	Family name
P297	ISO 3166-1 alpha-2 code
P298	ISO 3166-1 alpha-3 code
P1160	ISO 4 abbreviation
P1813	Short name
P1559	Name in native language
P1477	Birth name
P1449	Nickname
P742	Pseudonym

Table A1.: Properties used in the Query together with their corresponding label

A.2. Candidate generation query

We use a single SPARQL query to generate all query candidates for a given set of entities. The following query is an example for $E' = \{$`Q937`, `Q13426745`, `Q47513150`$\}$:

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX wikibase: <http://wikiba.se/ontology#>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
PREFIX wd: <http://www.wikidata.org/entity/>
SELECT ?entity ?property ?template WHERE {
  {
    SELECT ?entity ?property (0 AS ?template) WHERE {
      {
        SELECT ?entity ?property WHERE {
          values ?entity { wd:Q937 wd:Q13426745 wd:Q47513150 }
          ?entity ?property ?x .
        }
        GROUP BY ?entity ?property
      }
      ?property_entity wikibase:directClaim ?property .
    }
  } UNION {
    SELECT ?entity ?property (1 AS ?template) WHERE {
      {
        SELECT ?entity ?property WHERE {
          values ?entity { wd:Q937 wd:Q13426745 wd:Q47513150 }
          ?x ?property ?entity .
        }
        GROUP BY ?entity ?property
      }
      ?property_entity wikibase:directClaim ?property .
    }
  }
}
ORDER BY ASC(?entity)
```

A.3. Relation index query

We use the following SPARQL query to get all relation aliases and the number of occurrences of each relation. With `ql:has-predicate` we make again use of a QLever-specific element which lists all relations that exist for an entity. [1]

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX wikibase: <http://wikiba.se/ontology#>
SELECT DISTINCT ?predicate ?alias WHERE {
{
        SELECT ?predicate WHERE {
            ?x ql:has-predicate ?predicate .
        }
        GROUP BY ?predicate
    }
    ?entity wikibase:claim ?predicate .
    OPTIONAL {
        ?entity @en@rdfs:label|@en@skos:altLabel ?alias .
    }
}
ORDER BY ASC(?predicate) ASC(?alias)
```

Bibliography

[1] T. Goette, “Simple question answering over wikidata,” *Master’s thesis at the University of Freiburg*, 2021.

[2] H. Bast and E. Haussmann, “More accurate question answering on freebase,” in *Proceedings of the 24th ACM international on conference on information and knowledge management*, pp. 1431–1440, 2015.

[3] N. Reimers and I. Gurevych, “Sentence-bert: Sentence embeddings using siamese bert-networks,” *arXiv preprint arXiv:1908.10084*, 2019.

[4] M. Yu, W. Yin, K. S. Hasan, C. d. Santos, B. Xiang, and B. Zhou, “Improved neural relation detection for knowledge base question answering,” *arXiv preprint arXiv:1704.06194*, 2017.

[5] P. Wu, S. Huang, R. Weng, Z. Zheng, J. Zhang, X. Yan, and J. Chen, “Learning representation mapping for relation detection in knowledge base question answering,” *arXiv preprint arXiv:1907.07328*, 2019.

[6] A. Bordes, N. Usunier, S. Chopra, and J. Weston, “Large-scale simple question answering with memory networks,” *arXiv preprint arXiv:1506.02075*, 2015.

[7] M. Petrochuk and L. Zettlemoyer, “Simplequestions nearly solved: A new upperbound and baseline approach,” *arXiv preprint arXiv:1804.08798*, 2018.

[8] D. Lukovnikov, A. Fischer, and J. Lehmann, “Pretrained transformers for simple question answering over knowledge graphs,” in *The Semantic Web–ISWC 2019: 18th International Semantic Web Conference, Auckland, New Zealand, October 26–30, 2019, Proceedings, Part I 18*, pp. 470–486, Springer, 2019.

[9] Y. Gu, S. Kase, M. Vanni, B. Sadler, P. Liang, X. Yan, and Y. Su, “Beyond iid: three levels of generalization for question answering on knowledge bases,” in *Proceedings of the Web Conference 2021*, pp. 3477–3488, 2021.

[10] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014.

[11] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need,” *Advances in neural information processing systems*, vol. 30, 2017.

[12] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” *arXiv preprint arXiv:1810.04805*, 2018.

[13] M. Honnibal, I. Montani, and S. Van Landeghem, “spacy: Industrial-strength natural language processing in python,” 2020.

[14] T. Mikolov, K. Chen, G. Corrado, and J. Dean, “Efficient estimation of word representations in vector space,” *arXiv preprint arXiv:1301.3781*, 2013.

[15] N. Prange, “Wikiquestions - a wikipedia-based factoid question dataset,” *Master’s project at the University of Freiburg*, 2019.

[16] M. Henderson, R. Al-Rfou, B. Strope, Y.-H. Sung, L. Lukács, R. Guo, S. Kumar, B. Miklos, and R. Kurzweil, “Efficient natural language response suggestion for smart reply,” *arXiv preprint arXiv:1705.00652*, 2017.

[17] R. Hadsell, S. Chopra, and Y. LeCun, “Dimensionality reduction by learning an invariant mapping,” in *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR’06)*, vol. 2, pp. 1735–1742, IEEE, 2006.

[18] H. Bast and B. Buchhold, “Qlever: A query engine for efficient sparql+ text search,” in *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, pp. 647–656, 2017.

[19] D. Diefenbach, T. P. Tanon, K. Singh, and P. Maret, “Question answering benchmarks for wikidata,” in *ISWC 2017*, 2017.

[20] M. Dubey, D. Banerjee, A. Abdelkawi, and J. Lehmann, “Lc-quad 2.0: A large dataset for complex question answering over wikidata and dbpedia,” in *The Semantic Web–ISWC 2019: 18th International Semantic Web Conference, Auckland, New Zealand, October 26–30, 2019, Proceedings, Part II 18*, pp. 69–78, Springer, 2019.

[21] X. Huang, J. Zhang, D. Li, and P. Li, “Knowledge graph embedding based question answering,” in *Proceedings of the twelfth ACM international conference on web search and data mining*, pp. 105–113, 2019.