MASTER'S THESIS

Simple Question Answering over Wikidata

Thomas Goette

Examiner:	Prof. Hannah Bast
Second examiner:	Dr. Fang Wei-Kleiner
Adviser:	Prof. Hannah Bast

CHAIR OF ALGORITHMS AND DATA STRUCTURES
DEPARTMENT OF COMPUTER SCIENCE
FACULTY OF ENGINEERING
UNIVERSITY OF FREIBURG

Freiburg, 2021

Declaration

I hereby declare that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.

I hereby also declare that my thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Freiburg, 14 October 2021

Thomas Goette

Abstract

Question Answering over Knowledge Bases (QAKB) aims to extract the answer to a natural language question from a Knowledge Base. We approach this task for the Knowledge Base *Wikidata* by translating a natural language question into its corresponding *SPARQL* query which returns the answer to the question. Our focus is on simple questions which means that the corresponding *SPARQL* query contains only one triple. We provide a modular, easy-to-extend QA pipeline and evaluate it on the *SimpleQuestionsWikidata* benchmark. With it, we also provide an evaluation web tool which enables analyzing evaluation results of the pipeline.

Contents

1. Introduction

Since the rise of the semantic web there is an abundance of knowledge available on the internet. Knowledge Bases (sometimes called Knowledge Graphs) like the discontinued *Freebase*[1] or its successor *Wikidata*[2] contain billions of general facts about the world. They predominantly use the Resource Description Framework (*RDF*) to store facts as triples each consisting of a subject, a relation (sometimes called property) and an object. The fact "The mother of Albert Einstein is Pauline Koch" could be formalized as a triple with the subject "Albert Einstein", the relation "has mother" and the object "Pauline Koch".

SPARQL (*SPARQL* Protocol And *RDF* Query Language) is a standardized query language for extracting information from Knowledge Bases. Essentially, *SPARQL* uses the same triples as *RDF* but allows us to use variables to specify the intent of returning a result set. The triples of the query are matched to the triples in the Knowledge Base, resulting in a set of possible values for the variables. Suppose we wanted to ask the question "Who is the mother of Albert Einstein?" We could use the following *SPARQL* query:

```
SELECT ?target WHERE {
  "Albert Einstein" "has mother" ?target .
}
```

There is one triple in the Knowledge Base with the subject "Albert Einstein" and the relation "has mother". Its object is "Pauline Koch" so that is the result of the *SPARQL* query.

One of the challenges when working with Knowledge Bases is that names like "Albert Einstein" are usually not unique. There might be several people with that name in the world and there might be a studio album and a painting with that name as well. *Wikidata* uses unique identifiers instead of names in order to remove such ambiguity. The famous scientist Albert Einstein is stored as `<http://www.wikidata.org/entity/Q937>` or `wd:Q937` for short. The "has mother" relation is stored as `<http://www.wikidata.org/prop/direct/P25>` or `wdt:P25` for short. The real query asking for the mother of Albert Einstein is therefore:[3]

```
SELECT ?target WHERE {
  wd:Q937 wdt:P25 ?target .
}
```

This means that we have to know the correct identifiers before being able to formulate a query. There is another difficulty: Let us assume we want to find all the cities that are part of *Wikidata*. Let us further assume we already know of the entity `Q515` (city)[4] and the relation `P31` (instance of) which gives its subject a class. We might formulate the following query:

[1] `https://developers.google.com/freebase/`

[2] `https://www.wikidata.org/`

[3] We omit the definition of the prefixes `PREFIX wdt: <http://www.wikidata.org/prop/direct/>` and `PREFIX wd: <http://www.wikidata.org/entity/>` in all *SPARQL* queries in this work.

[4] Outside of *SPARQL* queries, we omit the prefixes for entities and relations from now on.

```
SELECT ?city WHERE {
  ?city wdt:P31 wd:Q515 .
}
```

At the time of writing, the query yields around 9000 cities. However, the result is incomplete; it does not contain cities like `Q1010506` (Idaho Springs) or `Q64` (Berlin). The problem here is that these cities are not directly an instance of `Q515` (city), but only indirectly. They are in fact instances of `Q1093829` (city of the United States) and `Q200250` (metropolis). These special kinds of cities in turn are subclasses (`P279`) of `Q515` (city). As an additional complication, `Q1093829` (city of the United States) is a direct subclass of `Q515` (city), but `Q200250` (metropolis) is only a subclass of `Q1637706` (million city) which is a subclass of `Q1549591` (big city) which is finally a subclass of `Q515` (city). The correct query would therefore be:

```
SELECT DISTINCT ?city WHERE {
  ?city wdt:P31/wdt:P279* wd:Q515 .
}
```

It uses `/` and `*` to express a variable number of subclass relations and the keyword `DISTINCT` to remove duplicates from the result. (It returns more than 32000 cities.)

The above examples illustrate that extracting information from a Knowledge Base via *SPARQL* requires expert knowledge of both *SPARQL* and the internals of the Knowledge Base.

Several approaches have been taken in order to make the information from Knowledge Bases more accessible to not only experts, but everyone. For example, Bast et al. (2012) reduce complexity by introducing a graphical user interface (GUI) which aids the user in querying a Knowledge Base. Bast et al. (2021) suggest using intelligent autocompletion to make it easier to formulate *SPARQL* queries. However, the process of querying a Knowledge Base is still cumbersome and involves some knowledge of *SPARQL*.

A natural next step is to translate a natural language question like "Who is the mother of Albert Einstein?" or a request like "List all cities!" to the correct *SPARQL* query automatically. This is what we try to do in this thesis.

We restrict ourselves to so-called simple questions. Simple questions are questions which relate to a *SPARQL* query containing only one triple and one variable. Note though that no part of our approach is specific to simple questions so it should be easy to extend to more general questions in future work.

1.1. Problem definition

The task of Question Answering over *Wikidata* is: Given a natural language question q, find a *SPARQL* query c such that the intended answer for question q is the result of executing the *SPARQL* query c on *Wikidata*.

For the task of Simple Question Answering over *Wikidata* that we consider in this work, the query c is of the form `SELECT ?t WHERE {<body>}` where `<body>` is one triple pattern with the variable `?t` being either in the subject or in the object position.

Consider the question "What is the capital of Bulgaria?" The answer to the question can be found be executing the following query:

```
SELECT ?target WHERE {
  wd:Q219 wdt:P36 ?target .
}
```

(Q219 is "Bulgaria" and P36 is "capital of".) Following Bast and Haussmann (2015), we call the pattern of this *SPARQL* query ERT (Entity - Relation - Target), meaning that the variable is in the object position of the triple.

Now consider the question "Which books did J. R. R. Tolkien write?" It relates to the following query:

```
SELECT ?book WHERE {
  ?book wdt:P50 wd:Q892 .
}
```

(Q892 is "Tolkien" and P50 is the "author" relation.) We call the pattern of this *SPARQL* query TRE (Target - Relation - Entity), meaning that the variable is in the subject position.

Note that the TRE pattern is not necessary when working with *Freebase* because all data in *Freebase* is duplicated. *Freebase* stores both the fact that some book was written by a person and that a person wrote that book. In *Wikidata*, duplication is usually avoided[5] which makes both patterns necessary.

1.2. Contributions

- We provide a modular, easy-to-extend Python program (including a web API) which translates simple natural language questions into *Wikidata SPARQL* queries and returns their results using an external *SPARQL* backend. See chapter 3.
- We provide a web tool for visualizing and interactively exploring the evaluation results of said program. See chapter 4.
- We provide an evaluation of our program on the *SimpleQuestionsWikidata* dataset and an error analysis for various example questions. See chapter 5.

[5] There are exceptions. Some examples are P25 (mother)/P40 (child), P36 (capital)/P1376 (capital of) or the symmetric P26 (spouse) and P3373 (sibling) relations.

2. Related Work

A lot of research has been done on simple Question Answering over the Knowledge Base *Freebase*. For the last several years, the community has focused on Question Answering systems that rely heavily on deep learning.

Bordes et al. (2015) use a Memory Network which includes preprocessing and keeping in memory the entire Knowledge Base. It involves mapping both all facts from the Knowledge Base and the input question to an embedding space where a similarity score is computed. In order to reduce the number of candidates to rank, they perform a preselection with an entity linking step that is similar to ours. They also perform a transfer learning task by applying a system trained on *Freebase* facts to a different Knowledge Base called *Reverb*.

Dai, Li, and Xu (2016) use deep recurrent neural networks (RNN) and neural embeddings for a conditional factoid factorization. They first infer the relation that occurs in the question and determine the most likely entity based on the relation. However, they also limit the search space first by using an entity linking step similar to ours.

He and Golub (2016) use a character-level encoder-decoder framework with attention to first encode the question and then decode it to a tuple of form (subject, relation). They also use an entity linking step similar to ours in order to limit the search space.

Yin et al. (2016) use a different entity linking procedure which includes entities which are only partially matched in the question. They also combine both a character-level convolutional neural network (CNN) and a word-level CNN with attentive maxpooling for scoring candidates.

Instead of using a pipeline of specialized components, Lukovnikov et al. (2017) train a neural network in an end-to-end manner and leave all decisions to the model. They do this in order to avoid the construction of complex pipelines, in order to avoid error propagation from one pipeline component to the next and in order to simplify the reuse for a different Knowledge Base without manual adoption. They use both character- and word-level information in order to predict both the entity and the relation from the correct query. They employ an entity matching step similar to ours in order to limit the search space.

Yu et al. (2017) propose a hierarchical residual bidirectional LSTM model which maps both the question and the possible relations to an embedding which is compared with cosine similarity. They also use an entity re-ranking step between the first entity linking and the relation matching. The publication is the only one in this chapter which includes an evaluation on non-simple questions.

Mohammed, Shi, and Lin (2018) criticize the trend to make models more and more complex in order to gain only small improvements. They compare a multitude of combinations of different entity linking and relation prediction components each both using deep learning and not using deep learning. They report that deep learning has indeed advanced the state of the art but that strong baselines and less complex models achieve almost comparable results and should be explored further.

Huang et al. (2019) compute low-dimensional embedding representations for all entities and and for all relations of the Knowledge Base. After an initial limiting of the search space they use a bidirectional LSTM based on pre-trained word embeddings for inferring the embedding for the entity and the embedding for the relation occurring in the question. In a last step

they determine the fact in the Knowledge Base embedding space that best matches the entity embedding and relation embedding.

Wu et al. (2019) note a problem with the dataset used in most previous work. In particular, 99% of the relations in the test set also exist in the training data. Published approaches perform much worse if the number of unseen relations is larger as is the case for real-world applications due to the large sizes of Knowledge Bases. They extend the system by Yu et al. (2017) with an additional pre-trained embedding to solve the problem of unseen relations. They explore a different train/test split where 50% of the test set's relations are not part of the training data to evaluate their approach.

Other than all the mentioned works, Oliya et al. (2021) use *Wikidata* and evaluate on the same dataset as us. They use a Knowledge Base representation called sparse-matrix reified KB first introduced by Cohen et al. (2020).

3. Pipeline

The question goes through multiple steps of a pipeline. Our pipeline is based on the pipeline of Bast and Haussmann (2015). Accordingly, we call our program "Aqqu Wikidata". We use the NLP pipeline framework of spaCy by Honnibal et al. (2020) because of its modular nature and its flexible configuration possibilities.

In the first step, we perform tokenization on the question (section 3.1). In the second step, we find entities possibly occurring in the question (section 3.2). In the third step, we generate *SPARQL* query candidates (section 3.3). In the fourth step, we match the relations from the candidates to the question (section 3.4). In the fifth step, we rank the candidates by relevance (section 3.5). In the sixth step, we process the best-ranked candidates in order to present the answer to the question (section 3.6).

3.1. Tokenization

Tokenization is the process of splitting a given question into its tokens. A token is often the same as a word but there are exceptions. For example, contractions (like "I'm" for "I am") combine two tokens into one word. We use spaCy by Honnibal et al. (2020) for tokenization.

3.2. Entity linking

We try to find *Wikidata* entities in the question by matching the names of entities to the word n-grams in the question. We use an entity index for finding the names of entites (section 3.2.1). We explain the matching algorithm in section 3.2.2.

3.2.1. Entity index

In order to speed up the matching of question words to names of *Wikidata* entities, we create an inverted index beforehand. For that, we find all names related to every entity in *Wikidata*.

In particular, we use the following types of names: the label itself, the alternative labels (sometimes called aliases), the family name, the short name (e.g. "JFK" for John F. Kennedy), the name in native language (e.g. "Marshall Mathers" for Eminem), the birth name (e.g. "Robert Allen Zimmerman" for Bob Dylan), the nickname (e.g. "Barry" for Barack Obama), the pseudonym (e.g. "El Comandante" for Cristiano Ronaldo), the two- and three-letter ISO codes for countries (e.g. "DE" and "DEU" for Germany) and the ISO 4 abbreviation (e.g. "Proc. Natl. Acad. Sci. U.S.A." for the Proceedings of the National Academy of Sciences of the United States of America) (for the exact query see appendix A.1).

We first normalize the unicode representation of the names. We do this in order to circumvent unicode issues which might result in certain words not comparing equal even though they are.[1]

We then replace non-ascii characters by their closest ascii character. We do this in order to make it easier to match entities with unusual spellings. For example, the city "Lübeck" could

[1] For example, "ü" can be produced by "\xc3\xbc" or by "u" followed by "\xcc\x88". The two versions do not compare equal.

be hard to match for a person using an English keyboard (which does not have a key for the letter "ü"). Without this step, the entity could only be matched if an alias "Lubeck" existed. Our solution makes the matching independent of the existing aliases.

Finally, we turn all names to lowercase. We do this because people often ignore capitalization when typing in search forms.

Using the described process, "Lübeck" will become "lubeck" and "Frières-Faillouël" will become "frieres-faillouel".

The inverted index maps the processed names to their corresponding entity IDs. You can see a few samples from the entity index in table 3.1.

Table 3.1.: Samples from the entity index. It maps the processed names of all entities to its corresponding entity IDs.

Name	Entities
g.o.a.t.	Q41421[a], Q17090583[b], Q84357932[c], ...
hanseatic city of lubeck	Q2843[d]
his airness	Q41421[a]
lubeck	Q2843[d], Q55807847[e], Q41498755[f], ...
michael jordan	Q41421[a], Q1928047[g], ...

[a] Famous basketball player Michael Jordan
[b] Galveston Orientation and Amnesia Test
[c] G.O.A.T. (vocal track by Ndoe)
[d] Lübeck (city in Germany)
[e] Lubeck (locality in Australia)
[f] New Zealand politician Marja Lubeck
[g] German comics artist Michael Jordan

3.2.2. Entity linking

We go through all word n-grams of any length in the question, prepare them as described in section 3.2.1 (normalizing, replacing non-ascii characters and turning them to lowercase) and look them up in the entity index. If we find multiple possible entity matches (which is almost always the case), we sort them by the number of matched tokens in the question first and by their scores second. We keep the best $N_e \in \{10, 50, 500\}$ of them and postpone the decision on what is the best match to the ranking step.

Many QA systems use an externally trained Entity Linker. Compared to Named Entity Recognition (NER) and classical Entity Linking (EL) systems from NLP, our method has the advantage of being able to potentially match any entity in the KB without having seen it during training. We are depending only on the aliases in the *Wikidata* dataset. If an important alias is missing, we potentially cannot match the corresponding entity correctly.

The web API allows to skip the entire entity linking step altogether and instead provide the correct entity IDs together with the question as input. This is especially useful in combination with a frontend which enables the user to choose entities interactively, for example by using autocompletion[2] or a "Did you mean" functionality as detailed by Diefenbach, Hormozi, et al. (2017).

[2] See e.g. `https://github.com/ad-freiburg/aqqu-frontend`

3.3. Candidate generation

We generate *SPARQL* query candidates based on predefined patterns. Because we restrict our work to simple questions, we use only two patterns of *SPARQL* queries that both only use one triple. Even though we only use two simple patterns in this work, our approach is not specific to these patterns and should also work with more complicated patterns (see section 6.3 for suggestions).

Let E be the set of linked entities from section 3.2 and let R be the set of all relations that exist in *Wikidata*.

We generate ERT candidates like this: For each $e \in E$ we find all $r \in R$ such that (e, r, o) is a triple in *Wikidata* for any object o. We find all r by executing one *SPARQL* query for each e.

Similarly, we generate TRE candidates like this: For each $e \in E$ we find all $\hat{r} \in R$ such that $(s, \hat{r}, e)$ is a triple in *Wikidata* for any subject s. We find all $\hat{r}$ by executing one *SPARQL* query for each e.

The union of ERT candidates and TRE candidates forms the set of all query candidates.

3.4. Relation matching

The generated candidates use all kinds of relations that are part of some triple in *Wikidata*. They include relations which are not related to the question. We now check whether the relations from candidates match the input question in some way.

We run an external POS (Part-of-speech) tagger[3] on the question. We classify the POS tags[4] from table 3.2 as content tags and all others as non-content tags.

Table 3.2.: Content POS tags

Tag	Description
CD	Cardinal number
JJ	Adjective
JJS	Adjective, superlative
NN	Noun, singular or mass
NNS	Noun, plural
NNP	Proper noun, singular
RB	Adverb
VB	Verb, base form
VBD	Verb, past tense
VBN	Verb, past participle
VBP	Verb, non-3rd person singular present
VBZ	Verb, 3rd person singular present

For each candidate, we determine the tokens $t_1, t_2, \ldots, t_n$ in the question that

- were not matched to the entity of the candidate during entity linking (see section 3.2),
- are tagged with a content tag and

[3] We use the POS tagger from spaCy by Honnibal et al. (2020).

[4] The POS tagger uses the POS tags from the Penn Treebank tagset as listed at `https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html`.

- are not forms of the verbs “be”, “do” or “go”. (These do not contain much meaning.)

If the question is “Who is the mother of Albert Einstein?” and the tokens “Albert” and “Einstein” were matched to the corresponding entity, that would leave the tokens “Who”, “is”, “the”, “mother” and “of”. Of these, only the POS tags of “is” (VBZ) and “mother” (NN) are content tags. The token “is” is a form of the verb “be”, so t_1 is “mother”.

We determine all aliases $a_1, a_2, \ldots, a_m$ for the relation of the candidate (see appendix A.2 for the exact query to determine the aliases). We also consider these aliases with all stopwords[5] removed and call them $\hat{a}_1, \hat{a}_2, \ldots, \hat{a}_m$. We use the lemma (canonical form) for the words of the aliases (e.g. “be” for “were” or “mouse” for “mice”).[6] For example, if the alias a_1 of the relation `P25` (mother) is “is daughter of”, the lemmatized form $l(a_1)$ is “be daughter of”. The lemmatized alias without stopwords $l(\hat{a}_1)$ is “daughter”.

We compare the lemmatized forms of the tokens $t_1, t_2, \ldots, t_n$ with the lemmatized aliases. If $l(t_i) = l(a_j)$ for any i and j, we call that an exact alias match. If $l(t_i)$ is a substring of $l(a_j)$ for any i and j (like “married” is a substring of “married to”), we call that a contained alias match. If $l(t_i) = l(\hat{a}_j)$ for any i and j, we call that a no-stop match.

3.5. Ranking

The candidates are not ordered in any particular way. We want to determine how likely a candidate answers the input question and rank the candidates accordingly. The highest ranked candidate should lead to the correct answer to the question.

Let $\mathcal{C}$ be the set of generated candidates (see section 3.3). We project each candidate into a feature space. For that, we define a function $f\colon \mathcal{C} \to \mathbb{R}^{10}$ that maps a candidate $c \in \mathcal{C}$ to a vector containing feature values that describe the candidate. The ten elements of the feature vector are listed in table 3.3.

3.5.1. Rule-based ranker

We assign each candidate a score, to which the various features contribute with different weights.

To make the weights more easily tunable, we first rescale all feature values such that the values of one feature among all candidates lie between 0 and 1. Formally, let $f_j(c_k)$ be the jth element of the feature vector for the kth candidate c_k. Let $f_j^{\max} := \max_k(f_j(c_k))$ and $f_j^{\min} := \min_k(f_j(c_k))$. The function that maps a candidate to its rescaled feature values $\hat{f}$ is then defined by:

$$\hat{f}_j(c_k) := \frac{f_j(c_k) - f_j^{\min}}{f_j^{\max} - f_j^{\min}}$$

We use the following formula to calculate a final score s for every candidate c_k:

$$s(c_k) := 1000\hat{f}_{10}(c_k) + 100\left(\hat{f}_5(c_k) + \hat{f}_6(c_k) + \hat{f}_7(c_k)\right) + 10\hat{f}_2(c_k) + \hat{f}_1(c_k)$$

Intuitively, the formula expresses that the token coverage is the most important feature, followed by features related to relation matches, followed by features related to entities.

A high score coincides with high probability to match the users intentions, so we sort the candidates by their scores in descending order to get the final ranking.

[5] Stopwords are words that occur very often in a language and that do not contain much meaning. Examples for English are “and”, “the”, “also” and “by”.

[6] We use the lemmatizer of spaCy by Honnibal et al. (2020)

Table 3.3.: Features for candidate c

Name	Description
$f_1(c)$	**Entity popularity score** (number of sitelinks)
$f_2(c)$	**Entity label matches** (number of entities of the candidate that were matched to words in the question by label (as opposed to by alias))[a]
$f_3(c)$	**Number of entity tokens** (number of tokens in the question that were matched to entities)
$f_4(c)$	**Number of entity tokens (ignoring stopwords)**
$f_5(c)$	**Number of exact relation matches**[a b]
$f_6(c)$	**Number of contained relation matches**[a b]
$f_7(c)$	**Number of no-stop relation matches**[a b]
$f_8(c)$	**Number of relation tokens** (total number of tokens of the question that were matched to a relation in the candidate query; duplicates are only counted once)
$f_9(c)$	**Pattern complexity** (number of triples in the *SPARQL* query)[c]
$f_{10}(c)$	**Token coverage** (number of tokens of the question which are POS-tagged with a content tag and which are not some form of the verbs "be", "do" or "go", divided by the sum of feature $f_4(c)$ and feature $f_8(c)$)

[a] For simple questions, this is a binary value

[b] The different kinds of relation matches are explained in section 3.4

[c] For simple questions, this value is always 1. We include the feature only in preparation for future extensions with more complicated *SPARQL* patterns (see section 6.3).

3.5.2. Learned ranker

As an alternative to the rule-based ranker, we also learn a ranking of candidates based on an annotated dataset without any hard-coded scoring formula. For most ranking problems, there are usually many results that are relevant to the input query. Our ranking problem is different because one question is correctly answered by usually only one *SPARQL* query.

The obvious way to learn a ranking is to learn a score for each candidate and rank by this score. This approach would compare candidates of different questions. However, certain features might be good for one question (compared to the other candidates of that question) but bad when compared to candidates of a different question. To circumvent this problem, we use a pairwise-ranking approach. We transform the problem into a binary classification task in which we try to predict for two given candidates which one should be ranked higher.

Training

The training data consists of tuples $(q_1, g_1), (q_2, g_2), \ldots, (q_N, g_N)$ each consisting of a question q_i together with the gold *SPARQL* query g_i for extracting the result that answers the question. We first run the gold queries g_i on our *SPARQL* backend b (see section 5.1) in order to get the gold results $r_i := b(g_i)$.[7]

We disable the ranking step and run all N questions through the remaining pipeline. Let $c_{k,i}$

[7] The training data also contains gold results (as opposed to gold queries), but they are often incomplete and can even be outdated. In particular, the gold result contains exactly one element for every question, even when the true gold result contains thousands of elements. This is why we ignore the gold results from the dataset and use the gold queries instead.

be the kth candidate that is generated this way for question q_i. For every question, we determine the candidates for which the corresponding *SPARQL* query yields a result that matches the gold result for that question, that is we find the $c_{k,i}$ with $b(c_{k,i}) = b(g_i)$ (using set equality).

These candidates are considered to be correct. Any candidate for which the corresponding *SPARQL* query yields a result different from the gold result is considered to be incorrect. If more than 40% of the candidates for a question are correct in this sense, that question is ignored in order to reduce noise. This happens e.g. for questions asking for the sex or gender of a person because many of the incorrect candidates will return the same sex or gender as the correct candidate.

For each question q_i, we build candidate pairs $(c_{k,i}, c_{m,i})$ where $c_{k,i}$ is a correct candidate and $c_{m,i}$ is an incorrect candidate for the same question q_i. We do this by randomly sampling 200 candidates for every question (or taking all, if there are less than 200 candidates). Out of the sampled candidates, we ignore the ones which are correct. We combine every incorrect candidate $c_{m,i}$ from the sampled candidates with every correct candidate $c_{k,i}$ for question q_i to form a candidate pair $(c_{k,i}, c_{m,i})$.

We create two training samples from every candidate pair $(c_{k,i}, c_{m,i})$. The first sample is $((f(c_{k,i}) - f(c_{m,i})),\ f(c_{k,i}),\ f(c_{m,i})) \in \mathbb{R}^{30}$ (using flat vector concatenation). The corresponding label for the training sample is 1. The second is $((f(c_{m,i}) - f(c_{k,i})),\ f(c_{m,i}),\ f(c_{k,i})) \in \mathbb{R}^{30}$ with the label 0.

With these training samples we train a random forest (Breiman, 2001) containing 200 random trees.

Ranking

Let $\mathcal{C}$ be the set of generated candidates for the input question after relation matching (section 3.4). We create every possible combination of two different candidates (c_k, c_m) with $c_k \in \mathcal{C}$, $c_m \in \mathcal{C}$ and $k < m$. There are $n(n-1)/2$ such combinations.

The features for a combination of candidates (c_k, c_m) are given by $((f(c_k) - f(c_m)),\ f(c_k),\ f(c_m)) \in \mathbb{R}^{30}$. We predict labels $y_{k,m} \in \{0, 1\}$ for every such combination of candidates.

For the final ranking decision between c_k and c_m there are two cases: For $k < m$, we rank c_k higher than c_m if and only if $y_{k,m} = 1$. Otherwise we rank it lower. Accordingly, for $m < k$, we rank c_k higher than c_m if and only if $y_{m,k} = 0$. Otherwise we rank it lower.

This gives us the final ranking of the candidates. Note that because of the number of combinations of candidates, this step's running time is quadratic in the number of candidates to rank. The previous steps of the pipeline must make sure that the number of candidates to be ranked is low enough or there are efficiency problems.

3.6. Preparing the answer

Once the ranking is done, we throw away all but the best-ranked 200 candidates. We finally execute a slightly modified version of the *SPARQL* query for these best candidates on our *SPARQL* backend in order to get the answer to the original input question.

We use the `OPTIONAL` keyword in order to include the English label for the resulting entity if there is one. We also limit the size of the result set to 300. The true *SPARQL* queries for the two examples from section 1.1 look like this (the questions are "What is the capital of Bulgaria?" and "Which books did J. R. R. Tolkien write?"):

```
PREFIX wd: <http://www.wikidata.org/entity/>
```

```
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT DISTINCT ?result ?resultname WHERE {
  wd:Q219 wdt:P36 ?result .
  OPTIONAL {
    ?result rdfs:label ?resultname .
    FILTER (lang(?resultname) = "en") .
  } .
} LIMIT 300
```

```
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT DISTINCT ?result ?resultname WHERE {
  ?result wdt:P50 wd:Q892 .
  OPTIONAL {
    ?result rdfs:label ?resultname .
    FILTER (lang(?resultname) = "en") .
  } .
} LIMIT 300
```

If the result includes `?resultname`, we present that to the user as the answer. If it does not, we use `?result` instead. This is especially relevant for literals like numbers or dates which never have a label.

4. Evaluation tool

We provide a web tool for evaluating and analyzing the pipeline. It is split in two parts. The API is written in *Python*[1] and returns evaluation results in the json format. The frontend is written in *Vue.js*.[2] It calls the API in order to get evaluation data and visualizes it.

The web tool's most remarkable feature is the detailed question view. After choosing one evaluation run and one question that was evaluated in that run you are presented with the chosen question at first as can be seen in figure 4.1.

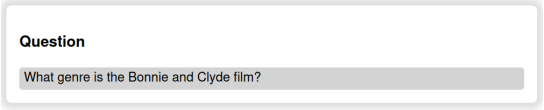


Figure 4.1.: Default view of the question when no candidate is selected

As can be seen in figure 4.2, the view also shows a list of the best-ranked candidates the pipeline generated. The candidates are selectable.

Figure 4.2.: View of the candidates for a question. It shows the total number of generated candidates, the number of candidates that survived the pipeline and the position of the first correct candidate. Underneath, there is a list of the best-ranked candidates.

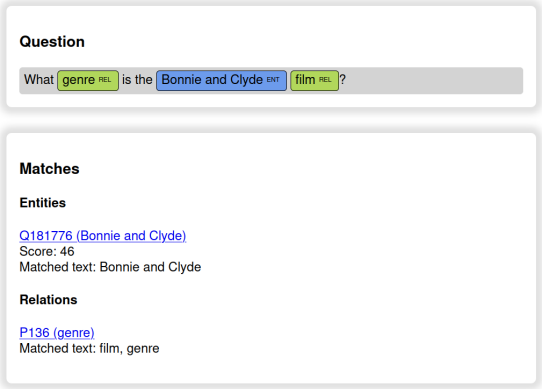
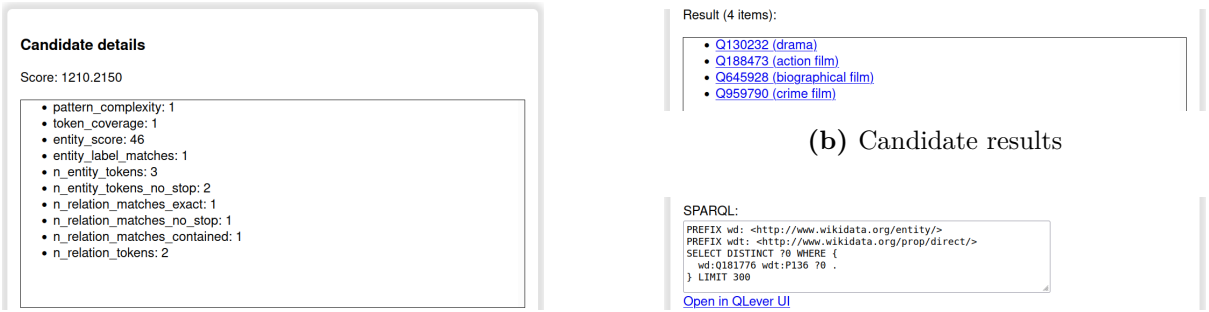


Figure 4.3.: View of the question when a candidate is selected. The matched entity and relation are highlighted in the question text and shown with additional information below the question.

If you select a candidate from the list, the questions view changes to include information about the entity and relation that were matched for the selected candidate. The updated view is shown in figure 4.3.

[1] `https://www.python.org/`
[2] `https://vuejs.org/`

As shown in figure 4.4, the view includes details about the selected candidate, in particular the candidate features (figure 4.4a), the candidate results (figure 4.4b) and the *SPARQL* query of the candidate (figure 4.4c).



(a) Candidate score and features

(b) Candidate results

(c) Candidate *SPARQL* query

Figure 4.4.: View of the candidate details

Independent of the selected candidate, the view also shows a list of entities that were matched during entity linking (see section 3.2). This can be seen in figure 4.5.

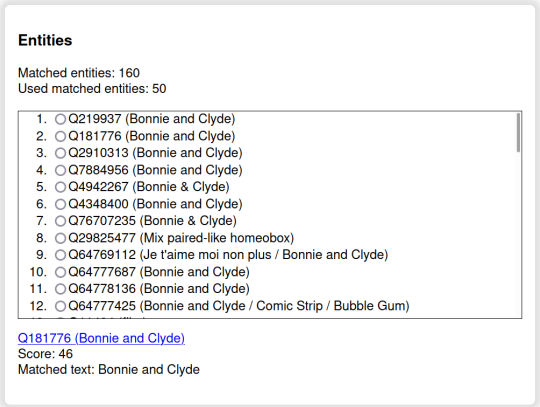


Figure 4.5.: View of the matched entities. It includes the total number of matched entities and the number of entities that were used for later steps of the pipeline. Below it, there is a list of matched entities with additional information for the selected one.

5. Evaluation

5.1. Hardware and *SPARQL* backend

Our experiments (including the precomputation of the indices) were performed on a PC with an Intel Xeon E5 v4 (4 cores) and 252 GB RAM. Note that 8 GB RAM are sufficient. The pipeline (and in particular the precomputation) requires 20 GB of storage space. The entire precomputation can be done in 2 hours.

We need to execute *SPARQL* queries in several parts of our pipeline and for the precomputation. We use the QLever engine first introduced by Bast and Buchhold (2017)[1] as our *SPARQL* backend. It supports a subset of the *SPARQL* 1.1 standard[2] and stands out by fast query response times. The *Wikidata* dump that QLever uses is from 2021-06-10 and contains 15.2 billion triples.

5.2. Dataset

We use the *SimpleQuestionsWikidata* dataset by Diefenbach, Tanon, et al. (2017) as a benchmark. We restrict ourselves to the subset which they classify as answerable on *Wikidata*. The training and test set contain 19481 and 5622 samples respectively. Each sample consists of a question together with the gold *SPARQL* query which answers the corresponding question.

The dataset is based on the *SimpleQuestions* dataset by Bordes et al. (2015). The original was created by human English-speaking annotators after automatically selecting facts from the Knowledge Base *Freebase*.

5.3. Evaluation results

We show a comparison of our results to results in the literature in table 5.1. Note that most authors evaluate by comparing the generated *SPARQL* query to the gold query. We instead compare the result that the generated *SPARQL* query yields to the result that the gold query yields. This difference in the evaluation method should be taken into account when comparing different methods.

Also note that Petrochuk and Zettlemoyer (2018) found the upper bound of the *SimpleQuestions* dataset to be 0.834. We do not know of similar findings about the *SimpleQuestionsWikidata* dataset.

Comparing the state of the art between the two Knowledge Bases, we see a drop of 10.5 percentage points when comparing *Wikidata* to FB2M and a drop of 6.7 percentage points when comparing *Wikidata* to FB5M. Note that Oliya et al. (2021) evaluate their system on both simple and non-simple questions which might contribute to their result being worse on only simple questions.

[1] The current version of the code can be found at `https://github.com/ad-freiburg/qlever`.
[2] `https://www.w3.org/TR/sparql11-query/`

Table 5.1.: Comparison with other QA systems. FB2M and FB5M are subsets of *Freebase* with 2M and 5M facts respectively. Results for them were evaluated on the original *SimpleQuestions* dataset. The other systems were evaluated on the newer and smaller *SimpleQuestionsWikidata* dataset using *Wikidata*.

QA system	Accuracy (FB2M[a])	Accuracy (FB5M[b])	Accuracy (*Wikidata*)
Bordes et al. (2015)	0.627	0.639	-
Yin et al. (2016)	0.683	0.672	-
Dai, Li, and Xu (2016)	-	0.626	-
He and Golub (2016)	0.709	0.703	-
Lukovnikov et al. (2017)	0.712	-	-
Yu et al. (2017)	**0.787**	-	-
Mohammed, Shi, and Lin (2018)	0.749	-	-
Huang et al. (2019)	0.754	**0.749**	-
Oliya et al. (2021)	-	-	**0.682**
Aqqu Wikidata (rules)	-	-	0.586
Aqqu Wikidata (learned)	-	-	0.564

[a] Subset of *Freebase* with 2M facts
[b] Subset of *Freebase* with 5M facts

We see that our rule-based approach cannot compare to the state of the art but comes surprisingly (difference of 9.6 percentage points) close considering there is no learning (and in particular no deep learning) involved.

Our learned ranker performs 2.2 percentage points worse than our rule-based ranker which is surprising. Unfortunately, we could not find the reason for this. We must assume an issue in our learning code.

We evaluate the influence of the parameter N_e controlling the number of matched entities that are used by the pipeline and show the results in table 5.2. We see that the results are almost identical for $N_e = 500$ and $N_e = 50$. The pipeline takes 1.5 seconds less on average for $N_e = 50$ but its average duration of 5.52 seconds is not quite usable in an interactive setting. Using $N_e = 10$, we see the R@1 value decrease by an absolute 4% (compared to $N_e = 50$) or 5% (compared to $N_e = 500$). However, the average duration per question decreases to an almost interactive 1.46 seconds. The results suggest that a value between 10 and 50 for N_e is a good idea, possibly combined with a more sophisticated entity linking component. None of the mentioned publications include their average query time so we cannot compare.

Note that the number of matched entities ranges from 7 ("who discovered 5551 glikson") to well over 40000 ("Which position in football did lee ho-jin play") which makes N_e an important parameter to ensure a consistent query duration.

5.4. Error analysis

We take a closer look at a few example questions that our rule-based pipeline answers incorrectly and analyze the reason for the incorrect result. We split the reasons into problems with the

Table 5.2.: Influence of the number of used entities N_e on the pipeline with the rule-based ranker. R@k is recall at kth position, meaning whether the correct result is part of the k best-ranked candidates. R@1 is the same as accuracy. AD is the average duration the pipeline takes for one query.

N_e	R@1	R@2	R@3	R@5	R@10	R@100	AD
500	**0.59**	**0.67**	**0.71**	**0.74**	**0.77**	**0.82**	7.09
50	0.58	**0.67**	**0.71**	**0.74**	**0.77**	**0.82**	5.52
10	0.54	0.66	0.69	0.72	0.75	0.77	**1.46**

pipeline, problems with the dataset and problems with *Wikidata*.

5.4.1. Problems with the pipeline

Incorrect entity match

Consider the question "who directed michael jackson: number ones". We identify the relation `P57` (director) correctly. Both the correct entity `Q5920952` (Number Ones) and the incorrect entity `Q2831` (Michael Jackson) lead to an almost identical feature representation. It differs only in the entity score which is higher for `Q2831` (Michael Jackson). In this case, the additional words "michael jackson" that are not part of the entity label are problematic for us. We rank the correct candidate as second.

In the question "What type of album is deliverin in?", the correct entity `Q5254062` (Deliverin') is not matched because of the missing apostrophe.

Relation not matched

Consider the question "which band made on earth to make the numbers up". We recognize `Q7090962` (On Earth to Make the Numbers Up) correctly, but do not match the correct relation `P175` (performer). None of its aliases ("artist", "musician", "played by", "portrayed by", "recorded by", "recording by", "dancer", "actor", "musical artist", "performed by", "actress", "sung by", "singer") are similar in text to "which band made".

The same problem occurs for the question "what killed tom held". The aliases of `P509` (cause of death) are "method of murder", "death cause", "die from", "murder method", "die of" and "died of", none of which match "what killed".

Another example is the question "in what category is hms e56 located" with `Q3720089` (HMS E56). The words "in what category is located" should match one of "ship class", "submarine class", "spacecraft class", "class of vessel" and "ship type", which are the aliases of `P289` (vessel class), but do not.

This problem is most prominent for demands like "Name a famous film director". They are often hard to match to a relation (`P106` (occupation) in this case).

Missing answer type differentiation

Consider the question "Where was Angela Merkel born?"[3] The word "born" matches `P1477` (birth name) via its alias "born as", `P19` (place of birth) via its alias "born at" and `P569` (date of birth) via its alias "born on". Without knowing the type of the result of a candidate and matching it to the question word ("where" in this case) there is no way of differentiating between the corresponding candidates.

Wrong pattern

For the question "What is a city within fort bend county, texas?", we match both `Q26895` (Fort Bend County) and `P131` (located in the administrative territorial entity) correctly. However, the position of the variable in the query is unclear (pattern ERT or TRE). The feature representations for the two corresponding candidates is identical so the order of the two candidates in the ranking is indeterministic.

Note that using the city constraint would make the decision between patterns easier but it would also make this a non-simple question (see section 5.4.2), more specifically of type "Single fact with type" (see section 6.3).

Either/or questions

Questions mentioning two possible answer options are hard because they contain many words that are not directly related to the entity or relation of the resulting query. This interferes with the matching process. One example for this problem is the question "is orpheus signed to mgm records or capital records". (From a logical standpoint, one could argue that the correct answer to the question could also be "yes", but that is besides the point.)

5.4.2. Problems with the dataset

Incorrect gold query

Consider the question "Who is the famous father of jack carter" with the following gold query:

```
SELECT ?target WHERE {
  ?target wdt:P40 wd:Q6111597 .
}
```

The query contains `Q611597` (Jack Carter) and `P40` (child) and returns both the father and the mother. The correct query (using `P22` (father) and returning only the father) is this:

```
SELECT ?target WHERE {
  wd:Q6111597 wdt:P22 ?target .
}
```

Non-simple questions

There are some questions in the dataset which are not truly simple questions. One example is the question "What is the name of a 1952 adventure film?" for which the gold query completely ignores the year mentioned in the question and returns all adventure films.

[3]Note that the question is not part of the dataset but there are examples in it with the same problem.

The question "what musician was born in garfield" is also not a simple question because it restricts the result both by place of birth and by occupation. The gold query ignores the occupation and returns all people born in Garfield.

Spelling mistakes

The dataset contains the question "what european country is fanny straw hair from" referring to `Q3576648` (Fanny Strawhair). There is also the question "what style of music does mariella farré sign". The pipeline does not deal with such spelling mistakes in any way.

Unresolvable Ambiguity

There are cases of ambiguity in the dataset that are impossible to resolve.

Consider the question "What position does Carlos Gomez play?" *Wikidata* currently contains four people with the name "Carlos Gómez" that each is part of a triple with the relation `P413` (position played on team / speciality).[4]

Another example is the question "what time zone is trenton located in?" There are four entities in *Wikidata* which are cities with the name of "Trenton".[5]

Without context, it is not possible for an automated system (or a human for that matter) to know which of the entities the question refers to.

5.4.3. Problems with *Wikidata*

Some questions reveal problems with the data of *Wikidata* itself. Missing data will probably be added at some point but incorrect data will always be a part of such a community project.

For the question "What is the name of a football player that plays as a forward", `Q55281700` (Elisa Gaspari) is incorrectly matched because of its alias "football Player" which should be a description instead.[6]

In the question "what language was used for the golden fortress", "the golden fortress" refers to the entity `Q22260787` (Sonar Kella). However, *Wikidata* does not contain any mention of this alternative name.

The question "Which country is the film sjors en sjimmie en de gorilla from" contains `Q2264085` (Sjors en Sjimmie en de Gorilla), but the entity has only a Dutch label and no English label in *Wikidata*.

The pipeline has problems with the question "who was the child of nefertari" only because someone changed the label of `Q210535` (Nefertari) to "Nefertarilllala".

[4]The four people are `Q203210`, `Q2747238`, `Q5750557` and `Q62592284`.

[5]The four cities are `Q25330`, `Q964707`, `Q482687` and `Q7838537`.

[6]Note that we have corrected the issues in this section in *Wikidata* so they should not occur if using a current dump.

6. Future work

Our simple approach is already successful on many simple questions. However, there are many possibilities to extend our approach and make it more useful both for simple and non-simple questions. In the following sections, we briefly suggest possible extensions. Due to the modular structure of our pipeline, extensions should be easy to integrate.

6.1. More sophisticated matching

In the previous chapter, we have shown numerous problems with the word-level matching. To improve, one should include character-level information for a more robust pipeline. Furthermore, one should not restrict oneself to text-based matching but include techniques like word embeddings for determining semantic similarity. The techniques used in most recent publications reflect this.

6.2. Unanswerable questions

The current pipeline answers almost every question because the rules for matching entities are rather lax. A step should be added which prunes candidates that probably do not relate to the question. This would enable the pipeline to return "NO ANSWER" to questions that cannot be answered with the data from *Wikidata*. Diefenbach (2018, section 3.4.4) even computes a confidence score at the end of his pipeline which, besides being used for pruning, tells the user a predicted probability of the best-ranked candidate being correct.

6.3. Additional query patterns and datasets

The restriction to simple questions is limiting the usefulness of the program. Some datasets exist which contain more complicated questions. We look at a few of them here in order to gather query patterns that should be supported in the future.

The question "Who was born in aguascalientes?" is part of the *SimpleQuestionsWikidata* dataset. They classify it as unanswerable on *Wikidata* because it is not a simple question on *Wikidata*. The question is answered by the following query:

```
SELECT ?target WHERE {
  ?target wdt:P19/wdt:P131* wd:Q79952 .
}
```

Besides `P19` (place of birth), it makes use of a variable-length relation path containing one or more `P131` (located in the administrative territorial entity) in order to include people who were born in e.g. `Q1150239` (Jesús María) which is a settlement located in `Q79952` (Aguascalientes).

We saw something similar in chapter 1 where the type of some cities was a subclass (`P279`) of `Q515` (city).

These are examples of transitivity in *Wikidata*. The concept of transitivity prevents some data duplication but as we have seen, it can make querying the data more complicated.

The LC-QuAD 2.0 dataset by Dubey et al. (2019) contains 30000 questions, their paraphrases and their corresponding *SPARQL* queries. The dataset is based on ten types of questions. We provide the corresponding *SPARQL* query for one example question of every type. Note that some of the types relate to multiple pattern variations depending on the position of the variables (just as ERT and TRE are variations of the single triple pattern).

1. Single fact
   This pattern is the one we cover with our ERT and TRE templates. We have shown several examples.

2. Single fact with type
   Example: "Billie Jean was on the tracklist of which studio album?"

   ```
   SELECT ?album WHERE {
     ?album wdt:P658 wd:Q193319 .
     ?album wdt:P31 wd:Q482994 .
   }
   ```

   The query contains `P658` (tracklist), `Q193319` (Billie Jean), `P31` (instance of) and `Q482994` (album). The pattern is similar to the Single fact pattern, but it also restricts the type of the result. Note that the mentioned transitivity problem occurs with this pattern.

3. Multi-fact
   Example: "What is the name of the sister city tied to Kansas City, which is located in the county of Seville Province?"

   ```
   SELECT ?city WHERE {
     ?city wdt:P190 wd:Q41819 .
     ?city wdt:P131 wd:Q95088 .
   }
   ```

   The query contains `P190` (twinned administrative body), `Q41819` (Kansas City), `P131` (located in the administrative territorial entity) and `Q95088` (Seville Province). In this pattern, the desired result is described by two general triples.

4. Fact with qualifiers
   Example: "What is the venue of Barack Obama's marriage?"

   ```
   PREFIX pq: <http://www.wikidata.org/prop/qualifier/>
   PREFIX p: <http://www.wikidata.org/prop/>
   PREFIX wd: <http://www.wikidata.org/entity/>
   SELECT ?place_of_marriage WHERE {
     wd:Q76 p:P26 ?s .
     ?s pq:P2842 ?place_of_marriage .
   }
   ```

   The query contains `Q76` (Barack Obama), `P26` (spouse) and `P2842` (place of marriage). It also uses prefixes which we have not mentioned so far, namely the `p:` and `pq:` prefixes. The object of the relation `p:P26` is not the spouse of Barack Obama itself, but an abstract statement node instead. (It takes the place of `?s` in this case.) By using the `pq:` prefix on the statement node, we can query information related to the spouse object, in this case, the place where the marriage took place.

5. Two intention
   Example: “Who is the wife of Barack Obama and where did he get married?”

```
PREFIX pq: <http://www.wikidata.org/prop/qualifier/>
PREFIX ps: <http://www.wikidata.org/prop/statement/>
PREFIX p: <http://www.wikidata.org/prop/>
PREFIX wd: <http://www.wikidata.org/entity/>
SELECT ?spouse ?place_of_marriage WHERE {
  wd:Q76 p:P26 ?s .
  ?s ps:P26 ?spouse .
  ?s pq:P2842 ?place_of_marriage .
}
```

   The query contains the same entities and relations as the last one. The `ps:` prefix can be used to get the object of the statement node which has the same effect as using the `wdt:` prefix directly. Also note that the query selects two variables (`?spouse` and `?place_of_marriage`) instead of one which corresponds to the fact that the question asks for two answers.

6. Boolean
   Example: “Did Breaking Bad have 5 seasons?”[1]

```
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
PREFIX wd: <http://www.wikidata.org/entity/>
ASK WHERE {
  wd:Q1079 wdt:P2437 "5.0"^^xsd:decimal .
}
```

   The query contains `Q1079` (Breaking Bad) and `P2437` (number of seasons) as well as a literal representing the number 5. Note that the query uses the `ASK` query form instead of the `SELECT` query form. This means that it does not return data from *Wikidata*, but instead checks whether the specified triple exists in *Wikidata* and returns only true or false. This corresponds to the fact that it is a yes/no question.

7. Count
   Example: “What is the number of Siblings of Edward III of England?”

```
SELECT (COUNT(DISTINCT ?sibling) AS ?count) WHERE {
  wd:Q129247 wdt:P3373 ?sibling .
}
```

   The query contains `Q129247` (Edward III of England) and `P3373` (sibling). It also uses the `COUNT` function to count the number of results.

8. Ranking
   Example: “what is the binary star which has the highest color index?”

```
SELECT ?star WHERE {
  ?star wdt:P31 wd:Q50053 .
  ?star wdt:P1458 ?color_index .
}
```

[1] The `ASK` query form is currently not supported by QLever.

```
ORDER BY DESC(?color_index)
LIMIT 1
```

The query contains `P31` (instance of), `Q50053` (binary star) and `P1458` (color index). It also uses the `ORDER BY` solution modifier to order the solutions and the `LIMIT` solution modifier to only return the first result.

9. String Operation
   Example: "Give me all the Rock bands that start with letter R"[2]

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
PREFIX wd: <http://www.wikidata.org/entity/>
SELECT DISTINCT ?band ?bandLabel WHERE {
  ?band wdt:P31 wd:Q5741069 .
  ?band rdfs:label ?bandLabel .
  FILTER(LANG(?bandLabel) = "en")
  FILTER(STRSTARTS(?bandLabel, 'R'))
}
```

   The query contains `P31` (instance of), `Q5741069` (rock band) and the `rdfs:label` relation which returns the label for an entity. It also uses the `FILTER` keyword with the `LANG` function to filter only English labels and the `FILTER` keyword with the `STRSTARTS` function to get only bands starting with the letter "R".

10. Temporal aspect
    Example: "With whom did Barack Obama get married in 1992?"[3]

```
PREFIX ps: <http://www.wikidata.org/prop/statement/>
PREFIX pq: <http://www.wikidata.org/prop/qualifier/>
PREFIX p: <http://www.wikidata.org/prop/>
PREFIX wd: <http://www.wikidata.org/entity/>
SELECT ?spouse ?date WHERE {
  wd:Q76 p:P26 ?s .
  ?s pq:P580 ?date .
  ?s ps:P26 ?spouse .
  FILTER(YEAR(?date) = "1992")
}
```

   The query contains `Q76` (Barack Obama), `P26` (spouse) and `P580` (start time). It also uses the `FILTER` keyword with the `YEAR` function to get only results with a start time in the year 1992.

The dataset was generated partly automatically and partly by the help of non-professionals. As a result, the quality of the dataset is lower[4] than the quality of smaller datasets created manually by professionals.

Task 4 of the QALD-7 open challenge by Usbeck et al. (2017) is called "Question answering over Wikidata" and provides a benchmark. It contains 150 questions with its respective *SPARQL* queries. The *SPARQL* queries are of varying complexity. The number of triples ranges

[2]The `STRSTARTS` function is currently not supported by QLever.
[3]The `YEAR` function is currently not supported by QLever.
[4]See `https://github.com/AskNowQA/LC-QuAD2.0/issues/4` for examples of issues with the dataset.

from one to five. *SPARQL* keywords that are part of the queries include `SELECT`, `ASK`, `COUNT`, `SUM`, `FILTER`, `ORDER BY`, `GROUP BY` and `HAVING`. Some queries contain a sub-query. Most of the queries only use direct relations (with the `wdt:` prefix) but there are some using statement qualifiers (with the `p:` prefix).

An example from the dataset is the question "Which mountain is the highest after the Annapurna?" with the corresponding gold query[5]:

```
SELECT DISTINCT ?uri WHERE {
  ?uri wdt:P31 wd:Q8502 .
  ?uri wdt:P2044 ?elevation .
  wd:Q16466024 wdt:P2044 ?elevation2 .
  FILTER (?elevation < ?elevation2) .
}
ORDER BY DESC(?elevation)
LIMIT 1
```

The query is far more complex than the simple queries we use. It is also fairly obvious that matching such a query is far more difficult as we cannot rely on only text matching certain labels.

The dataset is not free of errors. One question is "In which city did John F. Kennedy die?" They provide the following query as the gold query for the question:

```
SELECT DISTINCT ?uri WHERE {
  wd:Q9696 wdt:P20/wdt:P131 ?uri .
  ?uri wdt:P31 wd:Q515 .
}
```

The query uses `Q9696` (John F. Kennedy), `P20` (place of death), `P131` (located in the administrative territorial entity), `P31` (instance of) and `Q515` (city). However, the query yields no result. Yet again, the problem is the transitivity of cities of different sizes and qualities. `Q16557` (Dallas), the correct answer to the question, has the types `Q1093829` (city of the United States), `Q1549591` (big city) and `Q1637706` (million city) and is therefore only indirectly of type city.

Even though not applicable to this specific example, there is another error in the query if it should also be usable for other people. For Kennedy, `P131` works because *Wikidata* contains the fact that Kennedy died in a specific hospital which is located in a city. If we tried the same query for e.g. `Q9061` (Karl Marx), there would be no results because *Wikidata* only knows the city that Marx died in and the city is not located in another city. The correct query that also works for other people would therefore be:

```
SELECT DISTINCT ?uri WHERE {
  wd:Q9696 wdt:P20/wdt:P131* ?uri .
  ?uri wdt:P31/wdt:P279* wd:Q515 .
}
```

6.4. Scalability

For an interactive Question Answering service, a user expects query answer times of one second at most. Our system does not achieve such query answer times. Adding more *SPARQL* patterns (see section 6.3) will only make this problem worse.

[5]This query is currently not supported by QLever.

The bottleneck of the pipeline is the candidate generation process which requires live execution of many *SPARQL* queries. There might be ways to improve by combining multiple queries into one, by parallelization or by additional precomputation. For example, we could precompute both the types ("instance of") for every entity and the relations that usually exist for an entity of a given type. With that information, we could infer that e.g. the relation `P25` (mother) probably exists for `Q937` (Albert Einstein) because `Q937` is an instance of `Q5` (human).

6.5. Additional evaluation

The *SimpleQuestionsWikidata* dataset allows for a more thorough evaluation. In particular, since the dataset contains the gold query and thus the gold entity and gold relation for every question, it is possible to evaluate the entity linking step (see section 3.2) and the relation matching (see section 3.4) step individually. This can lead to a better understanding of what part of the pipeline should be improved and how. We have done so only for some examples in section 5.4.1.

6.6. Gold answer vs. gold query

We currently use the gold query from the dataset to determine the actual (and current) gold answer. We then compare the gold answer to the answer of candidates in order to determine whether the candidate is correct or not. This can of course be problematic: Let us assume the gold answer is "Germany". Many *SPARQL* queries lead to the answer "Germany" and are therefore labeled as correct, even though they might represent an entirely different question.

Alternatively, we could compare the gold query directly to the candidate queries. However, this is also problematic because there are questions which can correctly be represented by more than one *SPARQL* query.

An example of this is the question "Who is the child of Adele?" It can be answered by the query

```
SELECT ?target WHERE {
  wd:Q23215 wdt:P40 ?target .
}
```

with `Q23215` (Adele) and `P40` (child) or by the query

```
SELECT ?target WHERE {
  ?target wdt:P25 wd:Q23215 .
}
```

with `P25` (mother). It is almost a philosophical question whether the second query should be considered correct or not.

6.7. Spelling correction

Our pipeline currently cannot deal with spelling errors. For better usability in a real-world application, some kind of spelling correction or spelling robustness should be added. It should probably be integrated into the frontend for maximum usability.

A. Appendix

A.1. Entity index query

We use the following *SPARQL* query to find all the possible names for all entities.[1] We use the result to compute an inverted index for faster lookups as explained in section 3.2.

```
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 ?entity ?alias WHERE {
  ?entity wdt:P734?/(wdt:P297|wdt:P298|wdt:P1160|@en@wdt:P1813|
  ↪  @en@wdt:P1559|@en@wdt:P1477|@en@wdt:P1449|@en@wdt:P742|
  ↪  @en@skos:altLabel|@en@rdfs:label) ?alias
  ↪  .
  MINUS {
    # Ignore items internal to wikidata (around 7M)
    ?entity wdt:P31/wdt:P279* wd:Q17442446 .
  }
}
ORDER BY ASC(?entity) ASC(?alias)
```

You can see the used relations and their respective labels in table A.1.

Table A.1.: Types of names used for the entity index

Relation ID	Relation 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
skos:altLabel	Alternative labels
rdfs:label	Main label

[1] The query uses some QLever-specific syntax. The `@en@` restricts the results of the corresponding relation to only those of the English language and thus is a shortcut to a more complicated syntax involving `FILTER`s.

A.2. Relation index query

We use the following *SPARQL* query to find all the aliases for all relations.[1][2] We use the result to compute a lookup table for faster relation alias lookups as explained in section 3.4.

```
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 ?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)
```

[2]The QLever-specific `ql:has-predicate` relation lists all relations that exist for a subject.

Bibliography

Bast, Hannah and Björn Buchhold (2017). "QLever: A Query Engine for Efficient SPARQL+Text Search." In: *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*. CIKM '17. Singapore, Singapore: Association for Computing Machinery, pp. 647–656. ISBN: 9781450349185. DOI: 10.1145/3132847.3132921. URL: https://doi.org/10.1145/3132847.3132921.

Bast, Hannah and Elmar Haussmann (2015). "More Accurate Question Answering on Freebase." In: *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*. CIKM '15. Melbourne, Australia: Association for Computing Machinery, pp. 1431–1440. ISBN: 9781450337946. DOI: 10.1145/2806416.2806472. URL: https://doi.org/10.1145/2806416.2806472.

Bast, Hannah et al. (July 2012). "Broccoli: Semantic Full-Text Search at your Fingertips." In:

Bast, Hannah et al. (2021). *Efficient SPARQL Autocompletion via SPARQL*. arXiv: 2104.14595 [cs.DB].

Bordes, Antoine et al. (2015). *Large-scale Simple Question Answering with Memory Networks*. arXiv: 1506.02075 [cs.LG].

Breiman, L (Oct. 2001). "Random Forests." In: *Machine Learning* 45, pp. 5–32. DOI: 10.1023/A:1010950718922.

Cohen, William W. et al. (2020). "Scalable Neural Methods for Reasoning With a Symbolic Knowledge Base." In: *International Conference on Learning Representations*. URL: https://openreview.net/forum?id=BJlguT4YPr.

Dai, Zihang, Lei Li, and Wei Xu (Aug. 2016). "CFO: Conditional Focused Neural Question Answering with Large-scale Knowledge Bases." In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Berlin, Germany: Association for Computational Linguistics, pp. 800–810. DOI: 10.18653/v1/P16-1076. URL: https://aclanthology.org/P16-1076.

Diefenbach, Dennis (May 2018). "Question answering over Knowledge Bases." Theses. Université de Lyon. URL: https://tel.archives-ouvertes.fr/tel-02497232.

Diefenbach, Dennis, Niousha Hormozi, et al. (June 2017). "Introducing Feedback in Qanary: How Users Can Interact with QA Systems." In: ISBN: 978-3-319-70406-7. DOI: 10.1007/978-3-319-70407-4_16.

Diefenbach, Dennis, Thomas Pellissier Tanon, et al. (2017). "Question Answering Benchmarks for Wikidata." In: *Proceedings of the ISWC 2017 Posters & Demonstrations and Industry Tracks co-located with 16th International Semantic Web Conference (ISWC 2017), Vienna, Austria, October 23rd - to - 25th, 2017*. URL: http://ceur-ws.org/Vol-1963/paper555.pdf.

Dubey, Mohnish et al. (Oct. 2019). "LC-QuAD 2.0: A Large Dataset for Complex Question Answering over Wikidata and DBpedia." In: pp. 69–78. ISBN: 978-3-030-30795-0. DOI: 10.1007/978-3-030-30796-7_5.

He, Xiaodong and David Golub (Nov. 2016). "Character-Level Question Answering with Attention." In: *Proceedings of the 2016 Conference on Empirical Methods in Natural Language*

Processing. Austin, Texas: Association for Computational Linguistics, pp. 1598–1607. DOI: 10.18653/v1/D16-1166. URL: https://aclanthology.org/D16-1166.

Honnibal, Matthew et al. (2020). *spaCy: Industrial-strength Natural Language Processing in Python*. DOI: 10.5281/zenodo.1212303. URL: https://doi.org/10.5281/zenodo.1212303.

Huang, Xiao et al. (2019). "Knowledge Graph Embedding Based Question Answering." In: *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining*. WSDM '19. Melbourne VIC, Australia: Association for Computing Machinery, pp. 105–113. ISBN: 9781450359405. DOI: 10.1145/3289600.3290956. URL: https://doi.org/10.1145/3289600.3290956.

Lukovnikov, Denis et al. (2017). "Neural Network-Based Question Answering over Knowledge Graphs on Word and Character Level." In: *Proceedings of the 26th International Conference on World Wide Web*. WWW '17. Perth, Australia: International World Wide Web Conferences Steering Committee, pp. 1211–1220. ISBN: 9781450349130. DOI: 10.1145/3038912.3052675. URL: https://doi.org/10.1145/3038912.3052675.

Mohammed, Salman, Peng Shi, and Jimmy Lin (June 2018). "Strong Baselines for Simple Question Answering over Knowledge Graphs with and without Neural Networks." In: *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*. New Orleans, Louisiana: Association for Computational Linguistics, pp. 291–296. DOI: 10.18653/v1/N18-2047. URL: https://aclanthology.org/N18-2047.

Oliya, Armin et al. (2021). "End-to-End Entity Resolution and Question Answering Using Differentiable Knowledge Graphs." In:

Petrochuk, Michael and Luke Zettlemoyer (Oct. 2018). "SimpleQuestions Nearly Solved: A New Upperbound and Baseline Approach." In: *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Brussels, Belgium: Association for Computational Linguistics, pp. 554–558. DOI: 10.18653/v1/D18-1051. URL: https://aclanthology.org/D18-1051.

Usbeck, Ricardo et al. (2017). "7th Open Challenge on Question Answering over Linked Data (QALD-7)." In: *Semantic Web Evaluation Challenge*. Springer International Publishing, pp. 59–69. URL: https://svn.aksw.org/papers/2017/ESWC_2017_QALD/public.pdf.

Wu, Peng et al. (July 2019). "Learning Representation Mapping for Relation Detection in Knowledge Base Question Answering." In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Florence, Italy: Association for Computational Linguistics, pp. 6130–6139. DOI: 10.18653/v1/P19-1616. URL: https://aclanthology.org/P19-1616.

Yin, Wenpeng et al. (Dec. 2016). "Simple Question Answering by Attentive Convolutional Neural Network." In: *Proceedings of COLING 2016, the 26th International Conference on Computational Linguistics: Technical Papers*. Osaka, Japan: The COLING 2016 Organizing Committee, pp. 1746–1756. URL: https://aclanthology.org/C16-1164.

Yu, Mo et al. (July 2017). "Improved Neural Relation Detection for Knowledge Base Question Answering." In: *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Vancouver, Canada: Association for Computational Linguistics, pp. 571–581. DOI: 10.18653/v1/P17-1053. URL: https://aclanthology.org/P17-1053.