Language and Speech Technology Final Project Report

Boyd Belshof (s3012158)
Chiel Wijs (s3199886)
Jeroen van Brandenburg (s3193063)
Jördis Hollander (s2956543)

June 16, 2019

1 Introduction

A question-answering system (QA system) is a program that reads a question from a user, and attempts to automatically find the correct answer. The aim of this project is to develop a QA system in Python, leveraging SPARQL endpoints providing access to the Wikidata database. The system was tailored to the topic of music.

In this report we discuss the process of developing the QA system and showcase our results. First, we present the overall design and discuss the pipeline from receiving a question to determining its answer. Then, we evaluate the performance and accuracy of our program. Finally, we present a conclusion, improvements that could be made, and the division of labor between team members.

2 Design

Our question-answering system is built in Python and executes queries in SPARQL. The program reads a question from standard input and performs the following steps:

- classification
- extraction
- external querying

Each of these responsibilities is handled by a separate class, with the results being composed to make the QA system.

2.1 Previous Work

Before beginning with the bulk of the application design, it is useful to consider what was worked on in the previous assignments, and how this will have to be extended.

In the previous assignments we developed techniques for recognizing sentence structures using regex and spaCy, interacting with the Wikidata API and constructing SPARQL queries.

We determined during our previous work that different question types require different SPARQL queries (and maybe more than one as fallback strategies). We must ensure that the queries are able to capture answers in as broad a spectrum as possible using techniques to directly capture answers, determine possible property matches via SPARQL regex clauses, and also via the count clause.

The previous work did not require much in the way of explicit classification. However, due to the increased complexity of this assignment we must organize the various types of questions that may be classified. The use of classes to represent each question will nicely encapsulate the required data and allows each question to neatly hold its own answering logic. We make use of both regex and spaCy to create our classification rules.

Additionally, we need a method to simplify or normalize the input text to facilitate more robust creation of our classification rules. This is needed as the system grows in complexity as it encompasses more question types.

2.2 Classification

Different types of questions require different approaches to answering them. What data is relevant is distinct within the question types and affects both the extraction process and the queries that are required. Thus, classification is the first step our program takes when a question is entered into the terminal.

We decided to first normalize the input text to simplify the specification of the classifier. Our aim was to make the normalizer idempotent. To do this we use the industrial-strength natural language processor spaCy. Since some words are lemmatized incorrectly by spaCy, we implemented special rules to handle these cases. Then, the classifier uses this normal form to try to determine what type of question it is. This is done by looking for the specific structures of the four main question types.

The first question type is the descriptive question (e.g. "Who is Eminem?"). We noticed that these questions often start with the words "What" or "Who" and have a single concept as the subject of the question. If this is the case, we take a look at the second word, and check if it is the normal form of the verb 'to be'. For the final check, we test if there is an entity and property. We do this to split up the descriptive questions in two categories. If there only is an entity, we classify it as an `DescribeEntityQuestion`, if there only is a property, we classify it as `DescribePropertyQuestion`.

The second question type is the count question (e.g. "How many children does Jay-Z have?"). We assumed that these questions would be formulated using the expressions "how many" or "how much". As a result, classification involves simply checking if these are present. To prevent the propagation of errors, we also check if there is an entity and a property in the sentence, such that extraction is possible.

The third question type is the Boolean question (e.g. "Is Bono a performer?"). These type of questions always start with a verb. Therefore, this is our first criterion. We also check if there is an entity and property present in the sentence. We subsequently realized that our assumption does not hold for all questions. For example, "Name the members of Queen", begins with a verb but is not a Boolean question. The fix would have been simple to implement had this been realized sooner.

The forth and last question type are the "X of Y" questions (e.g. "What is the birth name of Eminem?"). These question types account for a bulk of the cases we considered. Thus, this is the question type we mainly focused on. These types often start with the words "who", "what", "where", "when" or "how". These different question types may necessitate different handling and thus they are represented as separate classes (still sub-classes of the main `abstract Question` class) for them (e.g. `XofYWhatQuestion`). Just like the other questions, we make sure there is an entity and a property in the sentence. The word *of* does not need to be present since we try to account for as much diversity of linguistic form of this question as possible.

If our program could not classify the question as one of these types (and logging is enabled),

we let the user know the question was unable to be classified and that the use of a different formulation is recommended.

2.3 Extraction

After the classification, we try to extract certain parts of the question, which are relevant for finding an answer. These parts are usually the property and the entity that will be searched for on Wikidata.

2.3.1 Extractor

The extractor class was created to normalize and prepare questions before searching for the corresponding Wikidata URIs. There were several cases where parts of the question were retrieved incorrectly by Wikidata which resulted in the return of incorrect URIs. These exceptional cases are handled in the `Extractor` class.

2.3.2 Thesaurus

Before retrieving the Wikidata URIs the program makes use of the `Thesaurus`, a dictionary that groups together words based on their meaning. This thesaurus is based on the list of questions sent in by all students following the Language Technology Practical course. This thesaurus is an addition to the already built-in searching functionality provided by Wikidata. This function looks for all URIs that could potentially match the given string. An addition to this functionality is necessary because a lot of different words can be used to describe the same thing. The entries in the thesaurus are cases in which we found that the built-in searching unsatisfactory. There are 47 word pairs in the thesaurus and it is built using a Python dictionary, a collection based on key-value pairs. An example of such a mapping is linking the key *members* to the value *has part* because band members can be found on the entity page of their band under the property *has part*.

2.4 Queries

2.4.1 P-values and Q-values

When the property and the entity are extracted from the question sentence, we map their corresponding P- and Q-values using the Wikidata search API via the `wbsearchentities` option. These values link to properties and entities on the Wikidata database. This is done with a SPARQL query in our class called `WikidataMapper`. In this class some handle some special cases are handled. For example, by default the Wikidata API responds to "Prince" as a member of the royal family instead of as the musician. Due to time constraints we simply hard-coded these special cases. By doing relevance/similarity analysis on the resulting options may alleviate the need for these hard-coded special cases.

2.4.2 Executing Queries

With the relevant P- and Q- values and the classification, the correct question class is constructed. Each question type hosts their own custom queries which are used in their strategies for answering the questions. In each case the queries are sent to Wikidata.

2.4.3 Primary and Fallback strategies

Each question type has its own strategy for finding an answer. The main approach is setup in the primary strategy however if no results were found, the fallback strategy gets a shot.

Both descriptive question types rely solely on the primary strategy because all they need to do is call the `WikidataMapper` on the entity or property and extract the description.

For count questions we have two different queries, one for the primary strategy and one for the fallback strategy. The primary strategy involves filtration of some properties and counting the resulting items (using SPARQL's `COUNT` clause). The fallback strategy deals with the case where one may directly search using a `number of *` property. In this case the number returned by the property should be used instead of the count of the results. For example using the first logic on the question "How many children does Bach have?" will result in the answer being 1. However, when using `number of` and returning the result directly we get the correct answer of 20.

For Boolean questions, we make a distinction between normal `BooleanQuestion`s and Boolean questions involving some form of the word "only". These `BooleanOnlyQuestion`s will be `True` if the property finds only one result. For the more generic Boolean question, it is true if there is any result. For example the question "Is Deadmau5 a composer" is `True` while "Is Deadmau5 only a composer"' is `False`. For `BooleanQuestion`s the fallback strategy is to simply return `False`. The logic here is that with no information we may presume that the property does not exist for entity. However, for `BooleanOnlyQuestion` we fallback to randomly choosing between `True` and `False`.

XofY questions consist of a common base class, `XofYQuestion`, and several sub-classes each of which represent the specialization of `Who`, `Where`, `When`, `What`, and `How`. Each of these sub-classes inherit their Wikidata query from the `XofYQuestion` superclass. The difference in the strategies for each of the specialized classes are in attempting to run their query for entities and properties relevant to the type of XofY question. For example, a when question will attempt to search for a *time of death*, whereas a where question will search for *place of death*.

2.5 Answer

Finally, the results that are returned by the query are printed in the console. For questions with multiple answers, the answers are separated by a tab. Boolean questions are answered with either 'Yes' or 'No'. When our program could not find an answer, it lets the user know by printing 'No answer found'.

3 Evaluation

3.1 Performance Measures

To evaluate the performance of the program, we were provided with 50 test questions. All of these test questions were in some way related to music. Each answer was graded n points where $n = \{0, 0.5, 1\}$. Zero points were awarded when an incorrect answer or no answer was given. One point was awarded for a correct answer. Half points were awarded for list-type answer for which at least half of the answer was correct.

3.2 Test Results

For twenty-nine out of the fifty test questions the program succeeded in retrieving an answer. Of these twenty-nine given answers twenty-two were correct. No half points were awarded to any list-type answers. This results in a score of twenty-two points out of fifty points possible. Out of twenty-nine given answers seven were thus incorrect. For twenty-one out of fifty questions the program failed to retrieve any answer.

3.3 Analysis

3.3.1 For Answered Questions

Classification of the incorrectly answered questions, which are seven, is as follows: one "X of Y", one "X of Y (when)", five "Boolean".

For the “X of Y” the wrong property was chosen. Before the entity was a naming of the type of entity, ’album’ in this case, which the program wrongly found as the property for the question.

For the “X of Y (when)” the correct property and entity were found but because of the wording the wrong P-number was retrieved from Wikidata.

For one of the “Boolean” questions the correct property, “singer-songwriter”, was found but normalization removed the “-” from the string. The normalized form could not be found on the Wikidata page, in contrast to the original form with the “-”. Two of these questions that were classified as “Boolean” where actually list-type questions. Both these questions were of the form “List the X of Y”. This mistake was made because we classify all questions that start with a verb as a “Boolean”. The remaining two “Boolean” questions were both of the form “Did X VERB Y?”. Here we would like to perform the query with “X” and “Y” as entity and attribute respectively. Instead, the program identified in both instances “VERB” as attribute.

3.3.2 For Unanswered Questions

There are various reasons as to why the program failed to retrieve an answer for twenty-one of the fifty test questions. Normalization can cause loss of information about the entity, for example “Red Hot Chili Peppers” is normalized to “Red Hot chili pepper”. This causes the program to look not at the band with said name as entity but at the similarly name food.

Sometimes wrong entities are found because the article “the” was removed. Most of the time “the” does not provide any useful information but, as the results show, there were instances where it was of crucial importance.

Faulty classification caused two of these questions to be unanswered, which is not surprising. The assumptions for the classifier were made to fit the most common form of all questions and, of course, not all questions are in their most common form.

Assumptions about how information can be retrieved from a classified question were also based on what we found to be the most common form of the question in case, the diversity of language obviously disagrees with standard forms. Because of a limited number of methods most unanswered questions were the result of faulty information extraction from the question, causing incorrect bits of text to be extracted as entity, property or sometimes attribute.

4 Conclusion

What makes language both so useful as a tool for communication and so difficult to work with is its diversity. Most questions can be formulated in a multitude of ways, making the information composing the question complex to identify and extract.

Determine the correct classification was complex and error-prone and would benefit from machine learning techniques. However, the system was able to correctly classify a significant amount of question formulations that we intended to cover.

Extraction has similar complications. However, if the classification were guaranteed (or assumed to be guaranteed), then the extraction has a more limited scope to deal with. However, our system does not handle nested questions well. Reformulation of our question hierarchy to include this is required with the necessary changes to the queries used, the question-solving strategies employed, and the extraction techniques needed.

We faced these problems and though are final system is relatively inaccurate, the experience has provided us a sufficient base to be able to further expand and improve any solutions we

implement in the space of natural-language processing and textual interaction.

5 Improvements

1. To make the classification more robust a neural net should be used. This net could be trained on large data-sets and would allow for the QA system to be more easily extended to other languages. The diversity in the formulation of the questions makes the formulation of handwritten rules difficult and prone to error. However, for the most part a person could classify the different question types at a glance. Automating this classification process may significantly streamline our pipeline.

2. Since we classify Boolean questions, by relying on the general property of the sentence starting of with a verb, we do not account for questions that are not of that nature also being classified as such, e.g. ("Name the partners of Madonna."). What could have been a more precise way of classifying a Boolean question is to assume that the normalized form of the leading verb of a Boolean question is the word *be* or *do*. This way one could differentiate between a list question, like naming the partners of Madonna, and a Boolean question, like Michael Jackson being a musician.

3. The thesaurus and the hard-coded exceptions in the `normalize_question` are a quick solution to a small scale QA system that wants to answer questions related to the topic of music. It accounts partially for not mapping the correct words. However in a larger scale QA system a more robust approach would be a similarity relevance analysis for the regarding topic. This would increase the chances of interpreting the correct URIs for the regarding entities and properties. For example in our system the mapping to Madonna would not output the singer but the religious figure. By using some form of relevancy scoring on the mapped results, we could select the most likely match. spaCy facilitates this but requires the use of larger datasets which significantly slow down the programs run-time.

4. The normalization should account for named entities and leave them as is. Improved detection of entities would also come with using one of the larger spaCy datasets, but again has incumbent run-time cost.

6 Attribution

Task	**Contributors**
Worked on report	Boyd, Chiel, Jeroen, Jördis
Generated test results	Chiel
Managed main application architecture	Jördis
Created classifier	Boyd, Jördis
Worked on Boolean classifier	Jeroen
Developed feature extraction	Chiel, Jeroen, Jördis
Developed Wikidata queries	Boyd, Jördis
Created `WikidataMapper`	Chiel, Jördis
Worked on edge-cases and `Thesaurus`	Boyd, Chiel, Jördis
Created question answering strategies for `Question` sub-classes	Jördis